TAMPERE UNIVERSITY OF TECHNOLOGY

STANISLAV MUHAMETSIN

A LANGUAGE-INDEPENDENT AGENT ARCHITECTURE

Master of Science Thesis

Examiners: professor Tapio Elomaa
assistant professor Jari Peltonen

Examiners and topic approved in the Faculty of Computing and Electrical Engineering Council meeting 19.8.2009

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
MUHAMETSIN, STANISLAV: A Language-Independent Agent Architecture
Master of Science Thesis, 49 pages, 7 Appendix pages
October 2011
Major: Software Science
Examiners: Professor Tapio Elomaa and assistant professor Jari Peltonen
Keywords: language-independence, software agent, software integration

In a software environment composed of multiple collaborating applications, an agent architecture is a suitable option for an integration architecture. The reason for this is that functionality requiring many applications as participants can be easily mapped to a single agent. Thus, maintainability increases, and functionality may be developed incrementally. However, the environment is often heterogeneous. The applications expose different APIs, potentially written in different programming languages. Therefore, an agent, during its lifetime, may need to use APIs written in different languages.

Most of the existing agent architectures are meant to be used in a single-language environment. Using such architectures directly in multi-language environment requires ad hoc solutions, which is undesirable. Expectedly, there are few agent architectures designed for multi-language environment. However, from the perspective of heterogeneous application integration, their approach is not feasible. Using application APIs written in different languages is cumbersome and prone to errors in these approaches.

This thesis presents a model for agent architectures aimed for heterogeneous application integration. The model allows mobile agents to use application APIs written in different languages in native way. The native API usage along with the mobility of the agents implies that agents must be transportable between applications written in different languages. Therefore, the full state of any agent is defined using formal methods to guarantee the interoperability between languages.

The agent architecture adhering to the presented model was implemented in two languages to validate the approach. Furthermore, the architecture is used in real-life software environment consisting of applications written in different languages. The native way of using application APIs is efficient and frees developers to concentrate on domain-related problems. However, integration of additional applications written in new language into existing software environment requires implementing agent architecture in that language. A benefit outweighing this extra work is increased maintainability. Since an agent is decomposed into tasks, they may be reused in other agents throughout the software environment.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
MUHAMETSIN, STANISLAV: Kieliriippumaton agenttiarkkitehtuuri
Diplomityö, 49 sivua, 7 liitesivua
Lokakuu 2011
Pääaine: Ohjelmistotiede
Tarkastajat: professori Tapio Elomaa ja yliassistentti Jari Peltonen
Avainsanat: kieliriippumattomuus, ohjelmistoagentti, ohjelmistojen integrointi

Uusia sovellusympäristöjä voidaan luoda integroimalla olemassa olevia sovelluksia. Usein uuden järjestelmän kokonaistoiminnan kannalta on tärkeää, että nämä ohjelmat tekevät yhteistyötä saumattomasti. Agenttipohjainen lähestymistapa soveltuu hyvin tällaisen sovellusympäristön integrointiin, sillä toiminto, joka vaatii monen eri ohjelman osallistumista, on helposti muunnettavissa agentiksi. Toiminnot voidaan toteuttaa asteittain, ja koska ne ovat loogisesti yhdessä paikassa, niistä tulee helposti ylläpidettäviä.

Yllämainitut sovellusympäristöt voivat kuitenkin olla heterogeenisiä, toisin sanoen integroitavien sovellusten paljastamat rajapinnat vaihtelevat ja voivat olla käytettävissä eri ohjelmointikielillä. Silloin agentti joutuu elinaikanaan käyttämään rajapintoja, jotka ovat tehty eri kielillä. Suurimmassa osassa nykyisistä agenttiarkkitehtuureista tätä asiaa ei ole kuitenkaan otettu huomioon. Niissä agenttiarkkitehtuureissa, joissa ympäristön heterogeenisyys on otettu huomioon, toisella kielellä tehtyjen rajapintojen käyttö on hankalaa ja virhealtista.

Tässä diplomityössä esitellään malli agenttiarkkitehtuureille, jotka soveltuvat hyvin heterogeenisen sovellusympäristön integrointiarkkitehtuureiksi. Esitellyssä mallissa agentit liikkuvat sovellusten välillä ja käyttävät sovellusten rajapintoja natiivisti. Kahden eri kielellä tehdyn sovelluksen välisen liikkuvuuden mahdollistamiseksi agentin tila määritellään formaalein menetelmin, jotka ovat riippumattomia mistään ohjelmointikielestä. Tämän lisäksi agentit jaetaan tehtäviin, jotka suorittavat agentin toiminnon pala kerrallaan.

Mallin toimivuuden osoittamiseksi mallin mukainen agenttiarkkitehtuuri toteutettiin kahdelle kielelle. Kyseinen arkkitehtuuri toimii oikean heterogeenisen sovellusympäristön integrointialustana. Sovelluksien rajapintojen natiivi käyttö oli sekä tehokasta, että auttoi ohjelmoijia keskittymään varsinaisten ongelmien ratkaisuun. Agentin jako tehtäviin edisti nopeata ja asteittaista toiminnallisuuden toteuttamista. Tehtävät voidaan myös uudelleenkäyttää toisissa agenteissa, joten ylläpidettävyyskin kasvoi. Haittapuolena tässä agenttiarkkitehtuurimallissa on se, että agenttiarkkitehtuuri joudutaan toteuttamaan jokaista sovellusympäristössä käytettyä kieltä kohden.

PREFACE

I would like to thank my colleagues for their professional support. I would also like to thank the examiners of this thesis, Tapio Elomaa and Jari Peltonen, for their excellent feedback for this thesis. Finally, I would like to thank my family for the support they provided during the writing process.

Tampere, Thursday 20th October, 2011

Stanislav Muhametsin

CONTENTS

TERMS AND DEFINITIONS

UML Unified Modeling Language [20]. The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of softwarebased systems as well as for modeling business and similar processes [20].

API Application Programming Interface [17]. An API is an interface exposed by a program allowing external programs or developers to access its functionality.

UTC Coordinated Universal Time. UTC is a time standard used to synchronize clocks around the world.

DCI Data, Context, Interaction [25]. DCI is a software design pattern aimed at maximizing extensibility and maintenance.

TCL Tool Command Language [3]. TCL is a simple, multi-paradigm programming language.

TCP Transmission Control Protocol [24]. TCP is the connection-oriented protocol most widely used on the Internet.

1. INTRODUCTION

A software modelling environment, called Trinity, is being developed in Tampere University of Technology (TUT) since 2008. It consists of multiple applications, each having some specific role in the environment. These applications let user visualize, manipulate, and manage models. An important part of the functionality in Trinity is *cross application functionality*, i.e., functionality that requires the coordination of many applications, while at the same time appearing to user as a continuous and single process.

To achieve cross application functionality, a way to integrate various applications is needed. In Trinity, an agent-based architecture is used as an integration platform for the applications. In this architecture, each cross application function is mapped to an agent, making such functionality easy to implement and maintain. The agents are also able to move among applications, and they can use the application APIs natively, i.e., in the same language as the API is written in. The native use of the APIs is efficient and enables developers to concentrate on the problems in their domain of expertise.

In 2009, Trinity expanded to include a new tool written in a different programming language than the rest of the applications. From the point of view of the agent architecture, this raised a problem on how APIs of the applications should be used in a multi-language environment. Additionally, the participants of cross application functionality may now be written in different languages. This means that the agent may need to be transported from one application written in some language to another application written in a different language. How should the agent architecture be designed in order to allow these kinds of scenarios not only in Trinity, but in heterogeneous application integration in general?

It has been noticed that agent architectures are suitable to be used as integration architectures [23, 6, 18, 15]. Despite this, only a few existing agent architectures are designed for an environment where language-independence is an important aspect. From the perspective of heterogeneous application integration, however, their approach is not feasible. This is because using APIs written in different languages is cumbersome and prone to errors in these architectures.

This thesis provides a model for the agent architectures aimed for heterogeneous application integration. The model preserves the benefits of agent architectures,

while making it possible to integrate applications written in different languages. The agents still use application APIs natively because of the benefits mentioned above. Additionally, the state of any agent in such model is defined using formal methods. This provides a sound theoretical basis strengthening the language independence of the presented model.

To validate the approach, the agent architecture based on this model was implemented in two languages. The implemented agent architecture is used in Trinity as the integration architecture. It supports language independence by making it possible to present a cross application function, requiring applications written in different languages, as a single agent. The implementation is generic enough to support cross application functionality in heterogeneous application environments other than Trinity.

This thesis is structured as follows. Section 2 provides background about different agent-based solutions, language-independent serialization mechanisms, and discusses a little about their implementation-specific details. It is followed by Section 3, which constrains and solidifies the requirements for the agent architecture with full support for language-independence. This section also introduces Trinity and its cross application functionality. In Section 4, the model for agent architectures designed for heterogeneous application integration is presented, along with language-independent data structures and serialization rules for them. Next, in Section 5, the specifications for implementations of the agent architecture in various languages are defined. The design and implementation specifications are evaluated in Section 6. Finally, the conclusions are presented in Section 7.

2. AGENT ARCHITECTURES AND LANGUAGE-INDEPENDENT SERIALIZATION

2.1 Background and Purpose of Agent Architectures

2.1.1 Overview of Agent Architectures

In the mid to late '90s a new concept for distributed applications emerged, called *mobile code application*. This concept means that applications are designed for a networked environment. In such an environment the code is no longer always available on one single machine. There are three different approaches described for designing mobile code applications: *remote evaluation*, *code on demand*, and *mobile agent*.[10] This thesis concentrates on the last approach.

Mobile agents are autonomous entities migrating between network places, using services provided by one or more of these places. During migration, the agents carry the logic on how to use the services and possibly some intermediate results.[10] Currently, there seems to be less applications using mobile agent paradigm, while the code on demand paradigm is very common [8]. However, as noted in [23], mobile agents are suitable approach for e.g. application integration. The advantages of the agents, like loose coupling and aspect-like nature, are also noted by Baumann et al. [6], Gray et al. [15], as well as Lange and Oshima [18].

Franklin and Graesser [12] provide the following definition for an *autonomous agent*:

> "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."

This definition emphasizes less the mobility and more the autonomy of an agent — the fact that an agent is aware of its environment and is temporal. Thus, all agents must have a beginning of their lifespan, and an end. Please note that the definition above does not rule out biological agents, such as humans. However, only agents in the context of software are considered whenever mentioned in this thesis.

Muhametsin et al. [19] explain the abstract concepts regarding *agent-based architectures*, from now on just *agent architectures*. The following text until Section 2.1.3

is rephrase from that paper. In that paper, agents are defined as "functional entities that use an infrastructure to achieve their goals". As a common terminology, the concepts of Execution Environment, Execution Unit, and Service are also defined.

The *Execution Environment* works as an abstraction for the operation environment for the agent, and provides ways to address Services and Execution Units. The *Execution Units* use the Services to provide the cross-application functionality. The *Services* may be API or data exposed by some application, or it may be a core service provided by Execution Environment. Typically, an agent is mapped to an Execution Unit. However, an agent may also consist of other agents or some other Execution Units. Using these concepts, a migration of agent happens among Execution Environments. During migration, the exchanged information is agent's execution state, the logic on how to use the services, and the intermediate results. If the Execution Environments are written in different languages, this information must be in a language independent form. [19]

2.1.2 Existing Agent Architectures

There is other aspect of the language independence in agent architectures besides the format of information to be exchanged in language independent agent migration. This aspect is the need of Execution Units to use Services written in different programming languages. There are at least three solutions to this aspect: Generic, Common Language, and Native.

In the *Generic* approach an interpreter or a core environment exists for all the needed languages. Additionally, the Generic approach also provides some generic mechanism to use Services from Execution Units, for instance, messages or events. In the *Common Language* approach, the Services are exposed through interfaces defined in a language usable by all Execution Units. Finally, the *Native* approach exposes Services directly in their native language. This requires ability to switch Execution Environments during the execution of the agent, and therefore, the problem of representing Execution Units, including an agent, in a language independent form.[19]

The various benefits and drawbacks of the three solutions mentioned above are discussed in [19]. The paper also discusses which solutions Agentscape [21], Ara [22], D'Agents [15] and TACOMA [16] use to solve the problem of using Services written in different programming languages. Additionally, the paper mentions an article by Cucurull et al. [9] discussing the interoperability of the complete agent architectures. The article discusses also inter-language interoperability resembling the Native approach, where different parts of the agent are implemented in various programming languages. This enables agent visiting and using Services written in various languages, and even dynamically selecting the most appropriate language to

use if there are many implementations in different languages for some part of the agent. As a disadvantage of this approach, the complexity of the agent is increased [9].

It is worth mentioning that none of the Agentscape, Ara, D'Agents, and TACOMA agent architectures use the Native approach. Therefore they do not need to solve the problem of language independent agent migration. Instead, they solve the problem of how the Execution Units use Services written in different languages.

2.1.3 Agent Architectures in Application Integration

When the thesis talks about *applications*, it is considered synonymous with *processes*. The key property of agent-based architecture used in application integration is the mobility aspect of agents. The mobility makes the agents natural candidates for performing goals spanning over multiple applications inside some software environments. The applications to be integrated may sometimes, though, be written in different languages, for the reasons out of the control of the developers. One such reason might be organization-wide policy to use some specific software in all possible situations.

While e.g. D'Agents has support for multiple languages, it is not possible to, for example, transport a `Java`-based agent and start it as a TCL process. The previously mentioned agent architectures (Agentscape, Ara, and TACOMA) are not answering to this question, but instead assume that the agent itself never actually visits an application written in different language than the agent itself. However, using large components via agents communicating with them in non-native way may become a performance bottleneck and complex to implement. Therefore language-independent agent transportation is an important aspect when designing agent architectures for heterogeneous application integration.

One of the key questions regarding the language-independent agent transportation is the transportation format. More concretely, how to represent data in language-independent fashion when it is inside an application, and when it is outside the application on some transportation channel, e.g. network.

2.2 Existing Mechanisms for Language-Independent Serialization

2.2.1 Extensible Markup Language

The most common technique for language-independent data exchange is *Extensible Markup Language* [27], or shortly *XML*. It is encountered mostly on the Internet, where Hypertext Markup Language 4.0 and higher is an application of XML [5].

Additionally, many applications use XML to store their settings. This way the same settings file can be used in different environments.

There also exists a *Document Type Definition* [27], *DTD* for short, mechanism for validation of XML files. The DTD provides ways to validate whether an XML document adheres to the set of the rules described in a DTD document. However, the content of XML document still remains same whether or not it is adhering to any DTD rules.

A short example of an XML document follows. The example document contains a string named *StringValue* with the value "StringVal", a number named *NumericalValue* with the value 667, and a complex value named *ComplexValue* which has two named sub-values, a string named *SubStringValue* with the value "SubStringVal" and a number named *SubNumericalValue* with the value 8. This example document is seen in Listing 2.1.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<documentRootElement>
  <StringValue>StringVal</StringValue>
  <NumericalValue>667</NumericalValue>
  <ComplexValue>
    <SubStringValue>SubStringVal</SubStringValue>
    <SubNumericalValue>8</SubNumericalValue>
  </ComplexValue>
</documentRootElement>
```

Listing 2.1: An example XML document.

The *<?xml* tag is required in the beginning of the document in order to parsers successfully identify the meta-information about the XML document. The *documentRootElement* is not part of the actual data to be encoded. It is there because multiple root elements are not allowed in XML.

2.2.2 JavaScript Object Notation

An alternative to XML is the *JavaScript Object Notation* [11], abbreviated *JSON*. It is a slightly more light-weight approach to language-independent data exchange, but because of that, it is not as extensible as XML. One current use of JSON is as an optional format for exporting bookmarks from Mozilla Firefox.

A major difference between XML and JSON is that where everything in XML is textual, JSON has certain pre-defined ways of expressing numbers, text, boolean values, value for nothing, arrays, and maps. As a short introduction to JSON, Listing 2.2 shows how the data shown in Listing 2.1 is expressed with JSON. Note that similarly to XML, JSON data must be inside a single construct.

```
{
  "StringValue" : "StringVal",
  "NumericalValue" : 667,
  "ComplexValue" : {
    "SubStringValue" : "SubStringVal",
    "SubNumericalValue" : 8 }
}
```

Listing 2.2: An example JSON output.

The output produced by JSON is slightly more compact than the corresponding XML document. Additionally, it is now clear that the value of *NumericalValue* really is a number. The value is not a complex construct, it does not have quotes around it, nor is it boolean value or value for nothing.

2.2.3 Apache Thrift

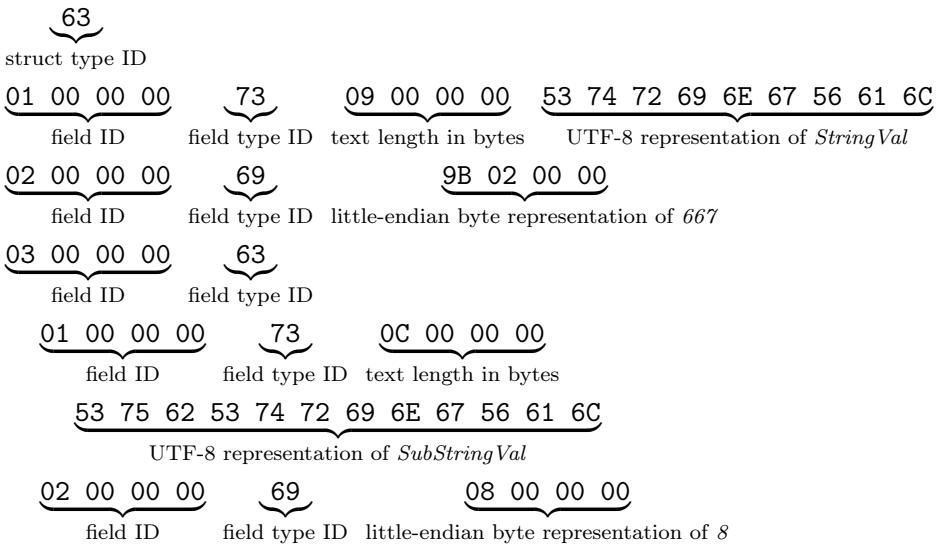
Apache Thrift provides a way to describe data and services, and is used by Facebook [26]. To describe data, Thrift provides *bool*, *byte*, *i16*, *i32*, *i64*, *double*, *string* as primitive types, and *struct* to be used in the same way as in C++. Instead of C++ types, however, the Thrift types are used (all primitive types, struct, as well as list, set, and map containers of any of the types). The serialization format for data is not specifically defined in Thrift, except that it should be deterministically serializable and deserializable, and the Thrift white paper [26] provides an API for writing meta-data before and after each data type, in addition to actually encoding the data itself.

Before listing the output of our example in the Thrift format, let us first define the data structure of our example. Listing 2.3 shows the structs to be used in order to serialize the data of our example. The map container is not suitable for having the *ComplexValue*, since the *ComplexValue* has members of different types: text and integer.

```
struct Example {
  1:string StringValue,
  2:i32 NumericalValue,
  3:ExampleSubStruct ComplexValue
}
struct ExampleSubStruct {
  1:string SubStringValue,
  2:i32 SubNumericalValue
}
```

Listing 2.3: An example Thrift of structs.

Using these structs, Listing 2.4 shows one possible output for serializing our example with Thrift. In this example, all integers are serialized in little-endian format, text has the amount of bytes of UTF-8 format as prepended meta-data, struct fields have field ID (32-bit integer) and field type ID (one byte) as prepended meta-data. The type ID for string is 73, the type ID for integer is 69, and the type ID for structs is 63.



Listing 2.4: An example of Thrift binary output.

2.2.4 Apache Avro

Apache Avro [28] is an API and transport protocol specification for serializing and deserializing data structures in a language-independent fashion. All serialization and deserialization must be done in accordance to an Avro *schema* in order to make compact binary output. In fact, the schema is *required* for each serialization and deserialization process. Similar to Thrift, Avro has a type system with primitive types being *null*, *boolean*, *int*, *long*, *float*, *double*, *bytes*, and *string*. The complex types of Avro are *record*, *enum*, *array*, *map*, *union*, and *fixed*. This rich type system is required due to the fact that a variety of data structures must be expressed. The schemas themselves may be represented in JSON format, and the schema for our example is seen in Listing 2.5. Note that the example is relatively simple. Avro schemas do provide more ways to define much more complex schemas.

```
{
  "type":"record",
  "name":"avro example schema",
  "fields":[
    {"name":"StringValue", "type":"string"},
    {"name":"NumericalValue", "type":"int"},
```

```
{"name":"ComplexValue", "type":{
  "type":"record",
  "name":"avro example subschema",
  "fields":[
    {"name":"SubStringValue", "type":"string"},
    {"name":"SubNumericalValue", "type":"int"}
  ]
}
}
}
```

Listing 2.5: An example of an Avro schema.

Avro has several tricks to reduce the size of the output serialization process. One of these optimizations is to transform a signed integer into an unsigned one by using ZigZag-encoding [1], and then serialize the resulting unsigned integer using a variable-length [2] format. Listing 2.6 shows the result of serializing the data content of our example into Avro binary format in accordance to the schema specified in Listing 2.5. The *VLZZ format* means Variable-Length ZigZag-encoded format.

```
        12          53 74 72 69 6E 67 56 61 6C
text length in VLZZ format   UTF-8 representation of StringVal
    B6 0A
667 in VLZZ format
        18          53 75 62 53 74 72 69 6E 67 56 61 6C
text length in VLZZ format   UTF-8 representation of SubStringVal
    10
8 in VLZZ format
```

Listing 2.6: An example of an Avro binary output.

2.3 Mathematical Notions and Implementation-Related Techniques

2.3.1 Mathematical Notions

The used mathematical notions and other notations are explained in this section. We start by defining the power set in Definition 2.3.1.

Definition 2.3.1 (Power set)**.** Let L be a set. The power set of L is $\mathcal{P}(L) = \{A \mid A \subseteq L\}$.

Basic notations for first-order logic are also presented. These notations are quite standard.

Definition 2.3.2 (First-order logic syntax shorthands)**.** Let $n \geq 1$. Then

1. a shorthand for $\forall x_1 \forall x_2 \forall \cdots \forall x_n \big(F(x_1, x_2, \ldots, x_n) \rightarrow G(x_1, x_2, \ldots, x_n)\big)$ is $\forall x_1, x_2, \ldots, x_n; F(x_1, x_2, \ldots, x_n) : G(x_1, x_2, \ldots, x_n)$,
2. a shorthand for $\exists x_1 \exists x_2 \exists \cdots \exists x_n \big(F(x_1, x_2, \ldots, x_n) \wedge G(x_1, x_2, \ldots, x_n)\big)$ is $\exists x_1, x_2, \ldots, x_n; F(x_1, x_2, \ldots, x_n) : G(x_1, x_2, \ldots, x_n)$,
3. a shorthand for $\forall x; x \in A : G(x)$ is $\forall x \in A : G(x)$, and
4. a shorthand for $\exists x; x \in A : G(x)$ is $\exists x \in A : G(x)$.

In order to operate on strings of symbols, some basic notations are defined for clarity. These consists of defining an empty string, and a set of all non-empty strings, of a certain alphabet.

Definition 2.3.3 (Basic definitions for operating with strings)**.** Let K be a set of symbols, and $n \geq 0$. Then

1. the basic finite string $b = a_1 a_2 \ldots a_n$, where $\forall i; 1 \leq i \leq n : a_i \in K$ is a *concatenation* of a_is, and
2. the length of b is $|b| = n$, and if $n = 0$, then $a_1 \ldots a_n = \varepsilon$, the *empty string*.
3. Additionally, $K^* = \{a_1 a_2 \ldots a_n \mid n \geq 0 \wedge \forall i; 1 \leq i \leq n : a_i \in K\}$ is a set of all *finite strings* of K, and
4. the set $K^+ = K^* \setminus \{\varepsilon\}$ is a set of all *non-empty finite strings* of K.

It is not always obvious what various terms mean in connection with *graph*s. We define the required terms now so that the definitions later in this paper would be unambiguous.

Definition 2.3.4 (Terms and definitions related to graphs)**.** Let V be a finite set of vertices, and $E \subseteq V \times V$ the set of edges. Then

1. (V, E) is a *directed graph*,
2. a *path* is a sequence $v_0 v_1 \ldots v_n$ such that $\forall i; 0 \leq i \leq n : v_i \in V$ and $\forall i; 1 \leq i \leq n : (v_{i-1}, v_i) \in E$, and
3. in order to say y *is reachable from* x, in other words that there exists a path from x to y, we use notation $x \rightarrow^* y$.

Please notice that Definition 2.3.4 (3) allows loops. There comes a need to express something to be optional. This is achieved using the special object **nil**.

Definition 2.3.5 (The **nil** object)**.** We define a special object **nil** to mean *something with no value*.

For example, when one wants to express optionality of some element x of some set X, one writes $x \in X \cup \{\mathbf{nil}\}$. When $x = \mathbf{nil}$, it is then interpreted as "having no value".

2.3.2 Data, Context, Interaction Pattern

The implementation of the agent-based architecture this thesis describes has been designed using the *DCI* (Data, Context, Interaction) [25] pattern. The concept of *data* contains relatively passive data model for the software, it should not contain any functionality or behaviour. The concept of *context* has the binding between each *role* and each object. One object may be in one or more roles — typically a role is an interface. The concept of *interaction* has the actual algorithms that operate on the objects through roles. The algorithms use the role map provided by the context to resolve the concrete objects based on the roles.

As a small example of what DCI pattern is aiming for, here is an excerpt from Rickard Öberg's weblog [7].

> "As mentioned, with the transition from procedural to OO we went from:
>
> `procedure(id,param1,param2)`
>
> to
>
> `object<id>.method(param1,param2)`
>
> And with DCI we are now going from:
>
> `service.method(id1,id2,id3,param1,param2)`
>
> to
>
> `context<id1,id2,id3>.interaction(param1,param2)`"

This thesis uses the following interpretation of the DCI pattern. The data is depicted by a standard UML [20] class diagram. Context classes or interfaces are marked with the *context* stereotype, and the used role types are positioned on the top-left side of the class or interface, with rectangles drawn in dashed line. Each context type holds one or more interactions, visible as methods. Typically, For all interactions of any context class there is a mapping available between the roles and objects, with roles as keys. This mapping is fully readable and modifiable. The role mappings are nested — any role mapping may have another role mapping as its parent. If exists, the parent role mapping is used to lookup objects when an object with a given role is not found in the current role map.

2.3.3 Composite Pattern

A data model often contains hierarchical data structures, which may be presented by a tree with arbitrary depth. For these kinds of data structures, a *composite* design pattern [13] is used. In this pattern, the tree items are either *nodes* or *leaves*. Nodes

may have other nodes or leaves as children. Leaves can not have any children. The composite pattern is shown in Figure 2.1.

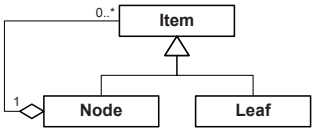


Figure 2.1: The UML class diagram of composite pattern.

Figure 2.1 shows that Item is a generalization of both Leaves and Nodes. This way the user of the hierarchical data structure may treat both nodes and leaves in a uniform manner [13].

2.3.4 Abstract Factory Pattern

When specifying interfaces to be used, the purpose is to abstract away the implementation. This means that we do not want the code that accesses the interface to access the implementation. However, in order to *create* the concrete resources implementing desired interfaces, there must be access to the implementation. To solve this problem, an *abstract factory* design pattern [13] is used. Figure 2.2 shows an example of an abstract factory design pattern.

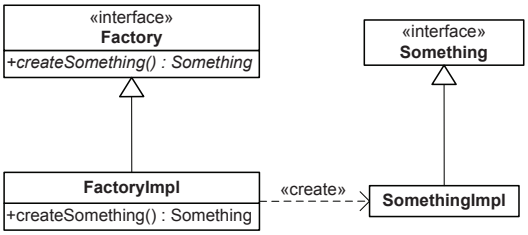


Figure 2.2: The UML class diagram of an example of an abstract factory pattern.

The abstract factory pattern enables code, that only has access to the interfaces, to create concrete resources. Additionally, there may exist several different implementations for a factory interface, and the user of the factory does not need to know about which implementation it is using.

3. AGENT ARCHITECTURES IN MULTILANGUAGE ENVIRONMENT

3.1 The Multilanguage Environment

3.1.1 Heterogeneous Application Integration

A special case of application integration is *heterogeneous application integration*. This means that applications within the software environment to be integrated are not alike, implying that they may be written in different languages.

The domain-specific functionality in any application integration is beyond the boundaries of single applications. This functionality requiring co-operation of multiple applications is called *cross-application functionality* in this thesis. Since the applications may be written in different languages, the execution state and serializing mechanisms native to some language can not be used.

Often architecture used in heterogeneous application integration is *message-based* [17, 30]. However, with message-based architecture solutions, the increase in number and complexity of cross-application functionality tends to directly decrease maintainability and extensibility of the applications. [30]

Agent-based solutions are optimized for solving integration of control flow. By using these solutions, each cross-application function can be implemented as a single agent. Thus the maintainability does not decrease with new functions. To further elaborate functionality of a heterogeneous software environment, a case study follows.

3.1.2 Case Study: Trinity Software Environment

A software modelling environment called Trinity has been developed at Tampere University of Technology since 2008. The environment is seen in Figure 3.1. All data related to modelling is kept in the *data repository*. The *Model Management Application*, or *MMA* for short, uses the repository to visualize the model structure, and each application of the *manipulator* "stereotype" uses the repository to visualize a model or part of it. Additionally, all manipulators and the MMA require the *Main Application* to run. The Main Application is responsible for some common repository-related features, which are beyond the scope of this thesis.

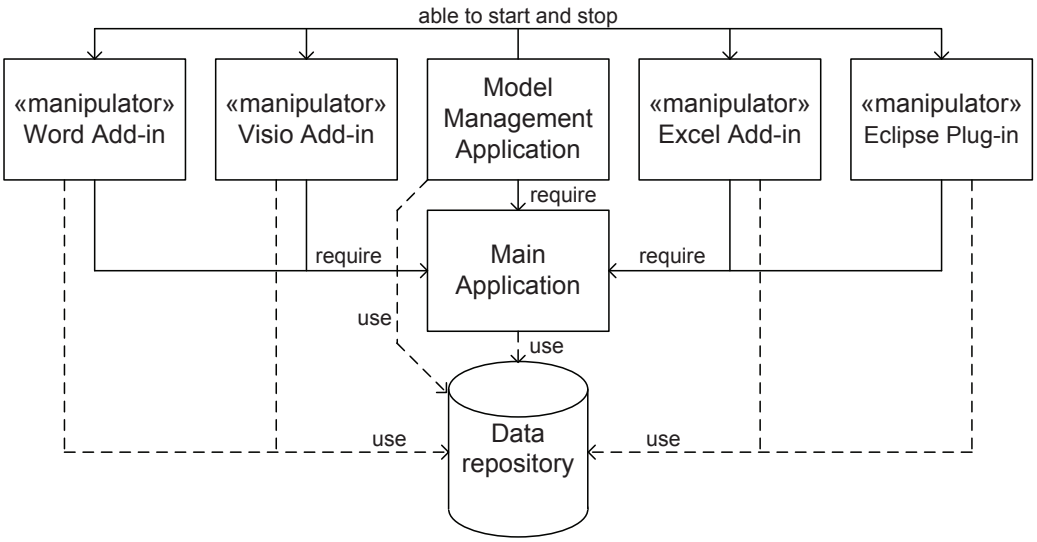


Figure 3.1: Overview of the Trinity modelling software environment.

The Eclipse plug-in is written in `Java` for compatibility with the Eclipse environment, while the remaining applications visible in Figure 3.1 are written in `C#` language. It is a high-level language with access to the APIs for programs in the Microsoft Office family (Word, Excel) and Microsoft Visio.

The data contained in the repository can be seen as a collection of models. The actual data schema of the repository covers much more than just that, but for the purpose of this thesis, it is sufficed to see the repository as a collection of models. The rest of unrelated data is composed of things and settings specifically related to some application part of Trinity, but it is not used by other applications.

Each model can have an arbitrary amount of *views*, while each view can have an arbitrary amount of *view elements*. Each view element contains the visualisation-related data, and has exactly one associated *model element*, which contains all data related to the modelling.

Trinity has multiple cross-application functions, each of them with constraints on where the functions may be started and what and where it does. Below is some of the current cross-application functionality of Trinity.

Function: Opening view

Possible starting applications: MMA, Main Application.

Description: commands a manipulator application to show a specified view stored in the repository. Will start up the manipulator application if it is not running.

Function: Closing view

Possible starting applications: MMA, Main Application.

Description: commands a manipulator application to close a specified view that it is possibly showing. Does nothing if the manipulator application is not showing that view.

Function: Closing manipulator application

Possible starting applications: MMA, Main Application.

Description: commands a manipulator application to shut itself down. Does nothing if that application is not running.

Function: Informing about opening a view

Possible starting applications: Any manipulator application.

Description: informs the MMA application that this manipulator application successfully opened a view. Does nothing if the MMA application is not running.

Function: Informing about closing a view

Possible starting applications: Any manipulator application.

Description: informs the MMA application that this manipulator application successfully closed a view. Does nothing if the MMA application is not running.

Function: Informing about manipulator application shutting down

Possible starting applications: Any manipulator application.

Description: informs the MMA application that this manipulator application is shutting down. Does nothing if the MMA application is not running.

Function: Selecting a view element

Possible starting applications: MMA, Main Application.

Description: commands a manipulator application to select a specified view element. If the view owning this view element is not currently open in this manipulator application, the application will open it.

Function: Validating a model

Possible starting applications: MMA.

Description: takes some model in Trinity repository, transforms it into an EMF [14] format, and uses Eclipse-based tool to perform validations on the model.

Part of the responsibilities of the MMA is to show which views are currently opened. However, views may be opened by other applications than MMA. Therefore the MMA needs to know which manipulator is showing which view. This is why the functionality related to informing MMA about opening and closing views is present.

3.1.3 An Example of a Cross-Application Function

Let us take opening view as an example functionality. This function is shown in Figure 3.2. The figure uses UML Statechart notation to represent each unit of work as a state, and transitions between states as transitions between units of work.

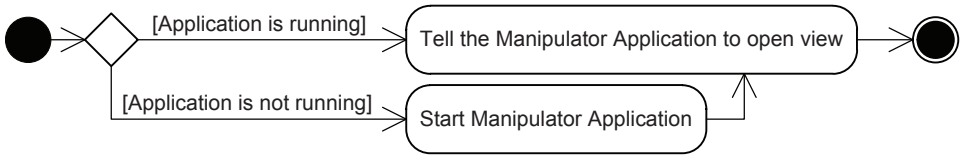


Figure 3.2: The cross-application functionality of opening view.

Figure 3.2 shows that the manipulator application will be started if it is not running. Only once it is running, will it be commanded to open the specified view. The function to open a view will have the view identification information and the exact type of manipulator specified at the beginning. This example will be continued after the design of the agent architecture has been specified.

3.2 Requirements for the Agent Architecture

3.2.1 General Requirements for the Agent Architecture

From the perspective of software development, the main goals for the language-independent agent architecture is to support incremental development and to increase maintainability. Additionally, the work load distribution is also essential requirement. In a context of heterogeneous application integration these requirements together allow quick implementation of complex cross-application functionality. This functionality still remains easily maintainable and evolves painlessly during software lifetime.

The requirement for an agent architecture with full language-independence with mobile agents is that the architecture itself must be designed in such a way that transporting agents between applications written in different languages is feasible. Therefore there must be a way to express the full state of an agent, in a language-independent way, so that the execution may be fully resumed after transportation. Since the execution states are incompatible between different languages, the execution model of the agent must be descriptive in a language-independent way. Additionally, to make the transportation transparent for users of the agent architecture,

there should be a concept of pause in execution flow, when the transportation may occur. During this pause, if the agent transportation occurs, the data and execution state of the agent are serialized and transported to another location. Consequentially, the agent architecture must have its own concept of execution state, so it would know where to resume the execution after transportation.

The agent architecture must have some specification on how the agents interact with an environment. This specification must be abstract enough to provide a uniform method of the interaction, but also be customizable enough to make interaction seamless when using the agent architecture.

The agent architecture must provide an easily understandable way for the applications to connect to each other. This connectivity specification must also be customizable, allowing users of the agent architecture to write their own implementations for non-standard network channels. In this thesis, the standard network channel is considered to be a TCP connection. This kind of connection easily abstracts away the connection either to a different application on the same computer, or a different application on a different computer.

Furthermore, the *description of the destination of agent transportation must be expressible in a language-independent format.* This is required, for example, in a situation where an agent is transported from application A to application B through application C. Each of these applications may be written in different languages. Therefore, the task destination lookup must operate on a more abstract level than the types of a programming language or other native constructs.

3.2.2 Constraints for the Agent Architecture from the Case Study

There are multiple approaches for designing a system to be used in a cross-language environment. For example, one approach would be to create a language-independent format to describe the behaviour and the knowledge necessary to use the components of the system. Another approach would be to develop a custom-made framework for each programming language in the environment, and agree on an interchangeable format for the frameworks to communicate and transport agents between each other.

The approach chosen for this use case is the latter, where agent architecture exists as a customized framework for each language, but the communication between the frameworks is of a certain format. This approach has been chosen for the Trinity environment because the Microsoft Office family only supports a few languages for implementation of Office add-ins. However, the plug-in for Eclipse must be implemented in `Java`, since Eclipse is written in `Java`. Therefore it is feasible for there to be many "incarnations" of the agent architecture on different languages.

There are two main requirements for the chosen agent-based approach to heterogeneous application integration. The first is that the execution state of the agent should be fully serializable in such a way that execution may be resumed in any language after deserialization of the agent. The second requirement is that the data state of the agent should be accessible and modifiable in a uniform way across many languages. The latter exists to ensure maintainability and extensibility of each cross-application function.

The most important requirement for the interchangeable communication format is that serializing and deserializing things with it does not happen too slowly. The agents may arrive in very quick succession in some applications, therefore it is useful to spend as few cycles as possible on (de)serialization. All performance- related resources that the application can spare should be given to the agents.

One can conclude from Section 2.2 that Avro is the most compact format of those presented in this thesis. The reason for Avro being the most compact is that the schema provides the required meta-information in order to know what type of data comes next. However, when creating a framework to be used in demanding environments, schemas are not always known for every single item being serialized or deserialized. Furthermore, in order to minimize workload, it is unwanted for an agent programmer to provide a schema every time he or she stores data into an agent. This is because the Trinity environment is meant to be developed in a rapid succession of brief steps.

In order to support serializing data without a schema, a *hybrid approach* is required. In this approach, the data is serialized in a more compact format whenever a schema is present. Otherwise the less compact format will be used in order to enable deserialization without the schema.

Error tolerance has little impact on the implementation of Trinity at this stage due to the reliability of the current network protocols. Thus the error tolerance is not considered as an important requirement, yet.

4. THE LANGUAGE-INDEPENDENT AGENT ARCHITECTURE

4.1 A Model for the Language-Independent Agent Architecture

4.1.1 Conceptual Architecture

As a starting point for the agent architecture, the design presented by Peltonen and Vartiala [23] was used and enhanced to fully support language-independence. A basis for this design was chosen to be Native approach, described in Section 2.1.2, mainly due to the fact that it is very efficient to call Services in native way. Figure 4.1 depicts this design, also presented in [19].

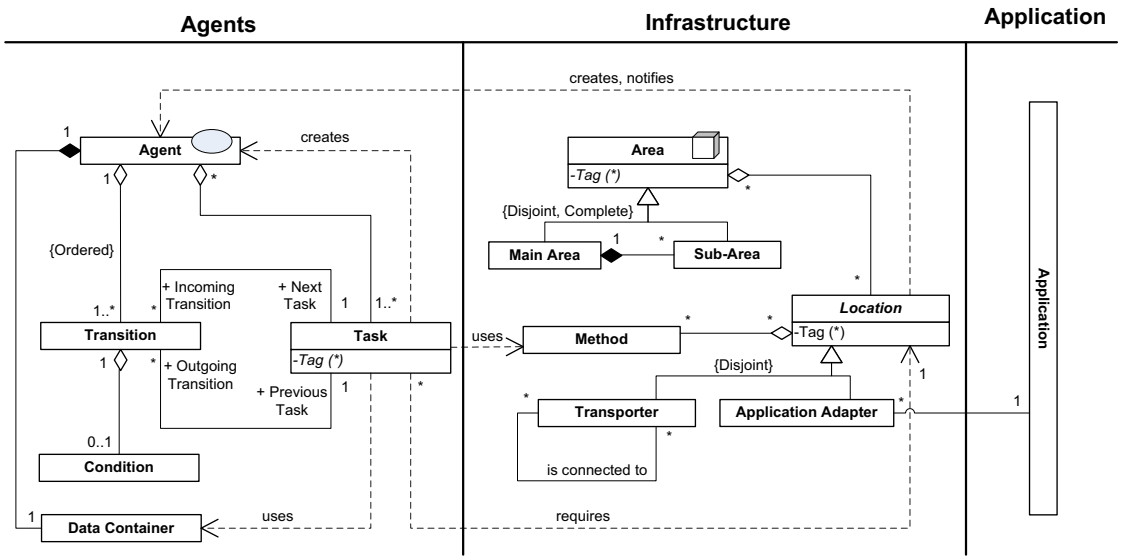


Figure 4.1: The design of the language independent agent architecture.

The following explanation of the Figure 4.1 until mentioned otherwise is a direct excerpt from [19]. In the proposed model, the agent infrastructure consists of hierarchical and parallel Execution Environments called *area*s. Each area is written in the programming language of its host applications. A *main area* can be composed of *sub-areas*. This way a larger Execution Environment, like a single computer in a

network, can be composed efficiently of smaller ones, having, e.g. their own process borders.

Areas consist of *locations* that provide Services, and can be either application adapters or transporters. An *application adapter* acts as a bridge between the agent architecture and an application. There may be several application adapters for an application, and each application adapter may expose only certain logically connected parts of the full API of the application. A *transporter* is a core service for transporting agents to and from other areas. Its purpose is to abstract away the actual implementation of transporting data over some network channel.

Agents are the top-level Execution Units in this model. An agent contains the business logic of a single aspect in the system. As an important feature, all agents are *mobile*, i.e. they are serializable and may be transported among different Execution Environments. Each agent has an execution plan composed of tasks, transitions between them, and conditions limiting the transitions. An agent has also a data container the tasks use to access the data carried by an agent.

Tasks are the *the smallest Execution Units* in an agent architecture based on this model. They use the Services provided by locations to perform their functionality. Each task is performed on a single location of a single area, and thus, a task is written in the same programming language as the location it uses. Within an agent, tasks can be executed sequentially, in parallel, or in a combination of both.

Each agent has a common *data container* for all its tasks. The data container stores the domain-specific data of the agent in a language independent format. This way the same data is available for all tasks, regardless of the language. Thus, the tasks can share data, i.e. communicate, with other tasks of the agent in an efficient and language independent way.

A critical part in a design of a language independent agent architecture according to this model, is the transportation of an agent. As mentioned before, each task is always performed in a single location in a single area. Therefore, the migration of the agent is constrained to occur only in the transitions between tasks.

Agents migrate between locations, and locations can be written in various languages. Therefore, the transportation format must be language independent. The need for transportation and the heterogeneous environment together imply that the full state of the agent, composed of *data state* and *execution state*, must be describable in a language independent fashion. However, the full internal execution state of the task does not need to conform to this requirement.

As seen in Figure 4.1, task, area, and location have *tags*. These tags exist because of the need for tasks to express the required location in a language independent way. Each tag is a single unit of some data describable in a language independent way. When a task needs some specific location, it only needs to provide the tags that

the location, and possibly the area owning the location, must have, instead of using native information. Using the tags provided by the task, the agent architecture can take care of finding the proper location, and transporting the agent there automatically. Importantly, also the tasks themselves have tags, since the native tasks need to be resolved based on language independent information of agent execution state. This enables the re-use of tasks, since an agent only needs to provide the tags for the required tasks, and the agent architecture will take care of looking up the correct task and transporting agent to the target site.

The transportation format used to transport agents and data between transporters needs to be carefully specified. If changes are made to the format, agent architectures for all languages used in a software environment must be upgraded to be compatible with the new format. If this kind of an upgrade needs to be done, is should be transparent for the users of the agent architecture, as they are not required to directly interact with the transportation of the agent.

At this point ends excerpt from [19]. As a side-remark, on high level, this implementation of agent architecture adheres to the DCI principles. The task acts as interaction, the data container acts as data, and the location together with area and methods act as context. This is purely coincidental and was not intended by the designers of agent architecture.

4.1.2 An Example of Implementing a Simple Function As an Agent

This section continues the example begun in Section 3.1.3. Constructs required by the function for opening view and related to the agent architecture are shown in Figure 4.2. For the purpose of demonstration, the manipulator in question is Eclipse. Therefore, the task to start a manipulator is written in `C#`, whereas the task to open a view is written in `Java`. The agent architecture infrastructure setup for this situation has one instance of Area for each application. The manipulator has a sub-area connected to the main area of the Main Application. Every time the manipulator starts, the sub-area is created inside the manipulator and connected to the main area. Consequentially, every time the manipulator shuts down, the sub-area is deleted in the manipulator and disconnected from the main area.

The functionality itself is represented by an agent. Each part of the function, shown in Figure 3.2 as a state, is a task. The agent will be created and started up in the area of the MMA, and the agent will be transported to the area of the manipulator for its final task. The task structure of the agent imitates closely the structure of the function itself. At the agent execution time, the task to start the

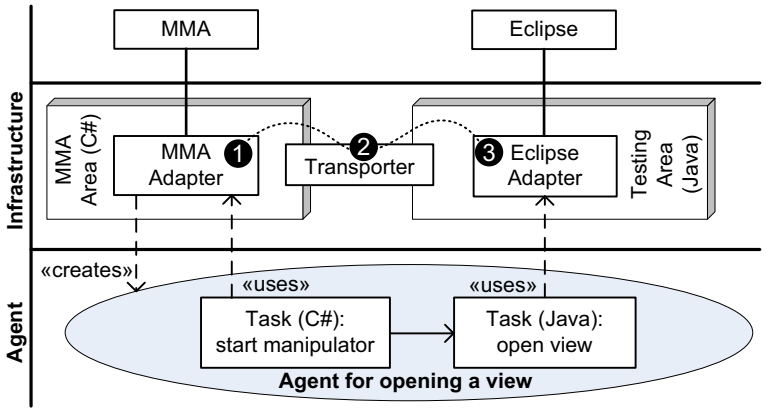


Figure 4.2: The constructs related to agent architecture of the example.

manipulator application will do nothing, if the manipulator is already running. The task to start the manipulator does not have any specific location requirements.

The tasks and locations are now neutral to what languages the MMA and the manipulator are implemented in. The tasks to start up the manipulator will be implemented in the language of the MMA, and the task to show data will be implemented in the language of the manipulator. The areas, locations, and transporters used will be implemented in whatever language the application they reside in are implemented. As an additional benefit, these tasks may be *reused* in other agents representing additional cross-application functionalities.

4.1.3 An Example of Implementing a Complex Function As an Agent

Complex cross-application functionality also maps into agent. Section 3.1.2 contains a list of various functionality in Trinity. Figure 4.3, adapted from [19], shows the agent and the infrastructure it requires for the last function in that list, the model transformation into EMF format. The "Modeling UI" in Figure 4.3 means same as MMA in this thesis.

As before, the dashed line represents the route an agent travels during the model transformation. At first, the agent is created as a result from user input in MMA. The agent then reads the desired model into its data container at stage two. Then, a language-independent agent transportation occurs at third stage. In stage four, the actual model transformation into EMF format happens. The transformation utilizes Eclipse API for creating EMF objects, and uses agent's data container as input. Finally, at stage five, the testing and validation tool is started and EMF model is passed to it. After the tool completes, it will display results to the user. More detailed description is provided in [19].

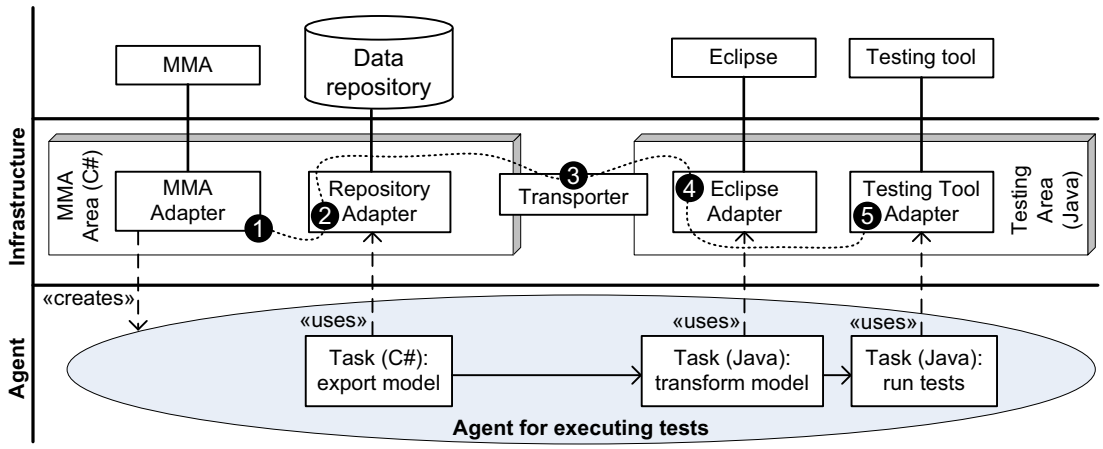


Figure 4.3: The applications, agent architecture infrastructure, agent, and tasks of the EMF model transformation.

4.2 Language-Independent Definition of the Agent State

4.2.1 The Formal Definition of the Agent State

The different implementations, one for each language, of the agent architecture presented in this thesis must be compatible between language boundaries. Therefore, the full state of any agent must be defined in a language-independent fashion. First the basic components used throughout definitions concerning agent state are defined in Definition 4.2.1.

Definition 4.2.1. Let the *set of all Unicode symbols* [29] be Σ. Then

1. the *set of all possibly usable task outputs* is $TO = \Sigma^*$,
2. the *set of all possibly usable instance IDs* is $I = \Sigma^+$, and
3. the *set of all possibly usable tags* is $T = \Sigma^+ \times \Sigma^+$.

A function to retrieve agent state, when agent's instance ID is given, is defined next. This function might return different results, depending at which point in lifecycle of agent the function is called, since the state of the agent most likely is changing as the agent is being executed. One way to map this function to reality is that this function is interpreted as the agent architecture. The function consumes the instance ID of the agent, and returns the state for such agent.

Definition 4.2.2 (The function to retrieve an agent state)**.** To begin, a set of all possible execution states for any agent is defined to be ES, and a set of all possible data states for any agent is defined to be DS. Let I be as in Definition 4.2.1 (2). The function to retrieve an agent state with given agent instance ID is $AS : I \rightarrow \{\textbf{nil}\} \cup (ES \times DS)$. AS produces **nil** if the agent with given instance ID does not

exist in agent architecture environment. However, if the agent exists, it will return the state of the agent $as \in ES \times DS$.

Next, constraints for ES and DS are defined, so that results produced by AS are meaningful. First, the constraints for a set of all possible execution states ES are defined.

Definition 4.2.3 (Constraints for execution states)**.** All elements of ES are of the form $(V, E, \hat{v}, VT, ET, EX)$, where

1. V is a set, and $E \subseteq V \times V$. Together V and E are interpreted as a directed graph. This graph is called the *agent task graph*, since every vertex represents a single task to execute, and edges are transitions between tasks.

2. $\hat{v} \in V$ is a *root vertex* of the task graph. It must hold that every vertex is reachable from the root vertex, that is, $\forall v \in V : \hat{v} \rightarrow^* v$.

3. VT is a function such that $VT : V \rightarrow \mathcal{P}(T)$, where T is as in Definition 4.2.1 (3), and $\forall v \in V : |VT(v)| < \infty$. The meaning of VT is that it returns a set of tags, which the agent task must have so that the task represented by this vertex gets executed.

4. ET is a function so that $ET : E \rightarrow \mathcal{P}(TO)$, where TO is as in Definition 4.2.1 (1), and $\forall e \in E : |ET(e)| < \infty$. The meaning of ET is that in order for the task represented by target vertex v_t to be executed, the task represented by source vertex v_s must return task output to_{vs} such that $to_{vs} \in ET(v_s, v_t)$.

5. $EX \in \mathcal{P}(EP \times ST)$ is a set of the *execution lifelines* of the agent. The EP is a set of all possible paths starting at $\hat{v}$ so that $EP = \{ep_0 ep_1 \dots ep_n \mid n \geq 0 \wedge ep_0 = \hat{v} \wedge \forall i; 1 \leq i \leq n : (ep_{i-1}, ep_i) \in E\}$, and ST is a set of execution statuses such that $ST = \{\textbf{executing}, \textbf{in\_transition}, \textbf{stopped}\}$. The meaning of EX is that it is a set of all current *execution lifelines* of the agent.

Additionally, the elements $V, E, \hat{v}$, VT, and ET are *fixed* for each agent. That is, given specific agent instance ID, they must always stay the same regardless of the call time of AS.

Each time the task completes, the vertex representing this task is added to the path of the current execution lifeline. Furthermore, for all ep_{i+1} in all paths, the task executed by ep_i returned task output to so that $to \in ET(ep_i, ep_{i+1})$. This means that execution path must traverse only those edges, for which there exist task output in ET.

If $EX = \emptyset$, then agent execution is considered to be stopped. If there are many execution lifelines with *status* other than **stopped**, then it means that the lifelines are executing *concurrently*. Next, the constraints for all possible data states DS are defined.

Definition 4.2.4 (Constraints for data states)**.** All elements of DS are of the form $(aid, hlid, N, VN)$, where

1. $aid \in I$, where I is as in Definition 4.2.1 (2), is the instance ID of the agent. It must hold that $AS(id)$ produces the state which has the instance ID aid of same value as id given to AS.
2. The $hlid \in I$ is the instance ID of the home location of the agent,
3. N is a domain-specific set of names for domain-specific data contained by this agent. Formally, $N \subseteq \Sigma^+$ and $|N| < \infty$.
4. VN is a function $VN : N \to LI$. It returns the language-independent data structure for each name in N. All elements of LI are software objects.

Additionally, the elements *aid* and *hlid* are *constants* for each agent.

4.2.2 Agent State During Transportation

In order to specify more concretely what sort of data is transported in agent transportation, a set of properties for agent state during transportation is defined. First, we decompose this state into two parts: the execution state, and the data state. Additionally, we assume that all areas, locations, and agents have an ID unique in the scope of their type. This means that an area and an agent may have the same IDs.

Data state — agent architecture related

Agent instance ID The ID of the agent.

Home location ID The ID of location, where agent was originally created.

History information An ordered list, in ascending order, of history information items. The order is determined by completion time of each task. After execution of a task has been completed, the history information item is appended to this set. This item is composed of the tags of the task, the ID of location, and an UTC time of completion of the task.

Destination

Criteria for area tags A set of tags that area must have in order to execute the next task. Optional.

Criteria for area ID The ID that area must have in order to execute the next task. Optional.

Criteria for location tags A set of tags that location must have in order to execute the next task. Optional.

Criteria for location ID The ID that location must have in order to execute the next task. Optional.

Criteria for agent history **True** if the location ID may be in history information of this agent. **False** otherwise.

Data state — domain-specific

Data container The contents of the data container of the agent.

Execution state

Task graph

Vertices A set of task graph vertices. Each vertex represents a task to be executed, and has an ID, unique in scope of this graph, and a set of tags. The actual task must have all the tags of this vertex to be executed.

Edges All the transitions between vertices. Each transition has the previous and next vertices, and the task output (a single string), which the previous task must return in order the transition to be used.

Next task The ID of the vertex in task graph, which will be used to search for the next task to execute.

In the execution state, there is only one vertex for the task to be executed next. This is because transportation occurs one execution lifeline at a time. Thus, only one task at a time may be executing at any point in the lifeline. There is a possibility for execution lifelines to perform *forks* (where a single lifeline spawns at least one new lifeline) and *joins* (where at least two lifelines merge into one). Hence, there is a need to define what exactly happens during such join. That is, what to do when two lifelines with different data container are merged? The answer for this question is not presented in this thesis, since the agent architecture is still in prototype stage.

4.3 Language-Independent Data Structures

4.3.1 Language-Independent Data Structure Types

In order to have a uniform way of handling the language-independent data in different programming languages, we define here a set of *language-independent data*

structure types. The types are further decomposed into *atomic types* and *composed types*. Each atomic type contains a single datum easily transformable and understandable in a type systems of most programming languages. Correspondingly, each composed type is a container for other language-independent data structures, and also easily transformable and understandable in a type systems of most programming languages.

Next we list all language-independent data structure types required for agent architecture operation. Formally, we define types for the software objects in set LI defined in Definition 4.2.4.

Atomic types

Int32 and Int64: data structures representing signed 32-bit and 64-bit integers, respectively.

Real32 and Real64: data structures representing 32-bit and 64-bit floating point values [4], respectively.

String: data structure representing a string of n characters, where $n \geq 0$.

Boolean: data structure having two possible values, either **true** or **false**.

Binary: data structure representing sequence of n binary numbers, where $n \geq 0$.

Composed types

List: data structure representing an ordered list of n other data structures, where $n \geq 0$.

Map: data structure representing an unordered list of named other data structures. Each name is unique in scope of a single Map.

These data structure types are enough to present data common to all programming languages. However, the user of the agent architecture is able to extend these types and provide her own custom types.

4.3.2 Serialization Rules for Language Independent Data Structures

Each language-independent data structure type has a serialization rule. This rule is *two-way* in the sense that unambiguous deserialization must be possible from the serialized format without any loss of data. The serialization format is binary in order to be as compact as possible. The rules have two variations: one variation should be used when schema is available during serialization of data structure, and the other

variation should be used when such schema is not available during serialization of data structure.

Now follows a list of serialization rules for all defined language-independent data structure types.

Int32 and Int64:

Serialized with schema: variable-length [2] ZigZag [1] encoding, in little-endian byte order.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing with schema.

Real32 and Real64:

Serialized with schema: 32 and 64-bit number, respectively, as defined in IEEE 754 [4], in little-endian byte order.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing with schema.

String:

Serialized with schema: amount of bytes of string encoded in UTF-8 format [29], as per serializing Int64 with schema, followed by the encoded string.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing with schema.

Boolean:

Serialized with schema: for **true**, byte 1, and for **false**, byte 0.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing with schema.

Binary:

Serialized with schema: amount of bytes, as per serializing Int64 with schema, followed by the bytes.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing with schema.

List:

Serialized with schema: amount of contained items, as per serializing Int64 with schema, then for each contained item, whether it matches any contained schema, as per serializing Boolean with schema, followed by, if needed, index of matched contained schema, as per serializing Int64 with schema, followed by serialized contained item.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing with schema.

Map:

Serialized with schema: for each *named sub-schema*, *in ascending alphabetical order*, whether the corresponding contained item is present, as per serializing Boolean with schema, followed by serialized contained item, as with List. For all remaining contained items, amount of the remaining contained items, as per serializing Int64 with schema, then for each contained item, the name of item in Map, as per serializing String with schema, followed by serialized contained item, as with List.

Serialized without schema: serialized unique string value for the type of data structure, followed by same value as when serializing remaining contained items when serializing with schema, only using all of contained items as remaining items.

The unique strings mentioned in the serialization rule list are listed in Table 4.1. It is assumed that possible extensions to the language-independent data structures will not be using these same strings for their custom data structure types.

Table 4.1: Unique strings for each data structure type defined in this thesis.

Data structure type	The unique string identifying the type
Binary	bi
Boolean	b
Int32	i
Int64	l
Real32	f
Real64	d
String	s
List	o
Map	m

5. IMPLEMENTATION OF THE LANGUAGE-INDEPENDENT AGENT ARCHITECTURE

5.1 The Implementation of the Agent Architecture

5.1.1 The Data Content Definition of the Agent Architecture

The implementation of the agent architecture follows DCI pattern quite closely. The data model of the agent architecture itself is very static, as seen in Figure 5.1. The only extension point is LIObject, covered later. The LIObject represents the base type of all language-independent data structures.

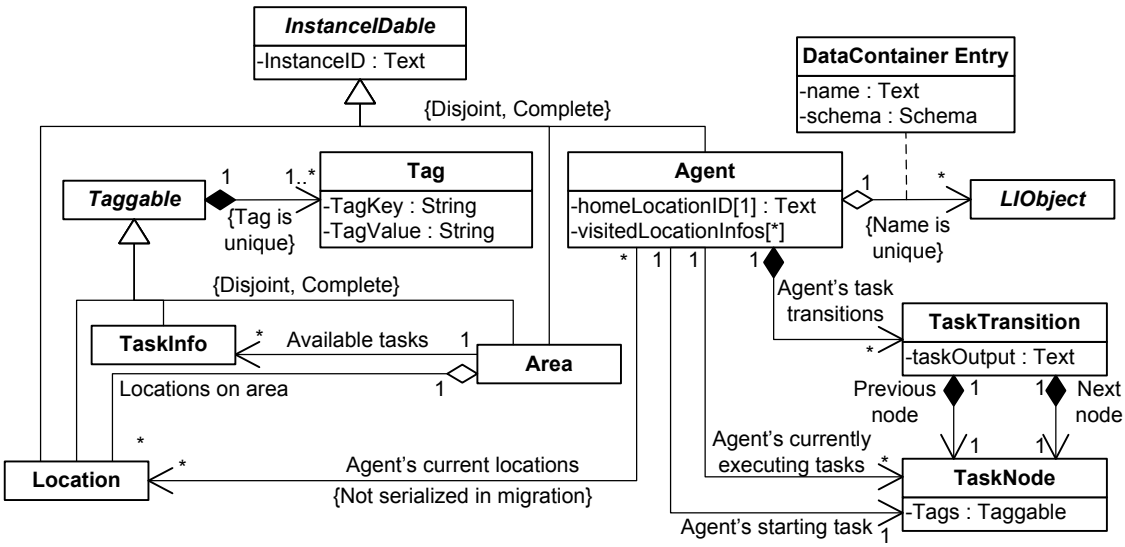


Figure 5.1: Data model for agent architecture.

The data model in Figure 5.1 shows that area, location, and agent all have instance IDs. This ID is some arbitrary string, and it should be unique in domain-specific scope, which may be anything from a single application on single computer to big cloud network. Area has zero or more locations, and zero or more tasks are available for execution on the area.

The agent contains an arbitrary amount of LIObjects, associated with mandatory unique name, and optional schema to use when serializing the LIObject. All

information about the tasks of the agent, and the order of tasks to execute, is in TaskTransition and TaskNode objects. The transition has previous node, next node, and task output. When task described by previous node completes with transition's task output, the task(s) described by the next node should be executed next. The information about currently parallelly executing tasks of the agent is available through association between Agent and TaskNode. The other association between those two classes specifies the very first task to be executed by the agent.

5.1.2 Functionality of the Agent Architecture

In order to retrieve the methods for location and tasks for tags, the DCI pattern is used. The contexts, interactions, and creators for location method and task retrieving are seen in Figure 5.2 and in Figure 5.3, respectively. In the Figure 5.2 there exists a singleton object LocationMethodRetrieverService, marked with *service* stereotype. The service contains the mapping used to look up the *context creator* for appropriate location based on the tags of the location itself and the area owning the location. This creator then creates an instance of the concrete subtype of the RetrieverContext containing the implementation for actually retrieving methods for this location.

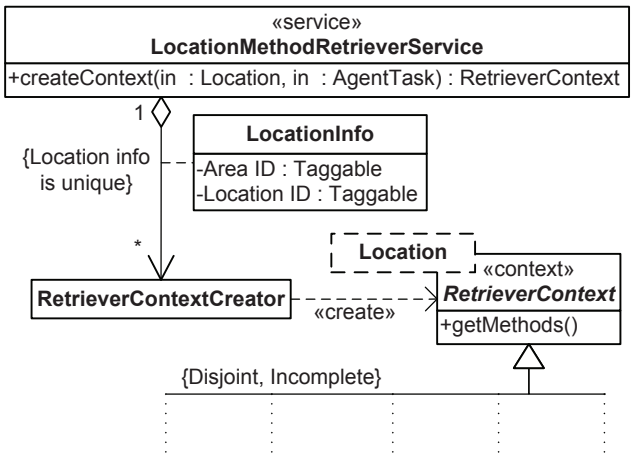


Figure 5.2: Contexts and interactions related to retrieving location methods in the agent architecture.

The Figure 5.3 has TaskPool as service mapping context creators based on task tags. After looking up the context creator, the service creates context, which is actually AgentTask in this case. The task has Agent and Location as two required roles. The task may use the location to retrieve the methods using LocationMethodRetrieverService. The agent architecture is responsible of maintaining the execution of agent and using TaskPool to retrieve correct tasks for the agent.

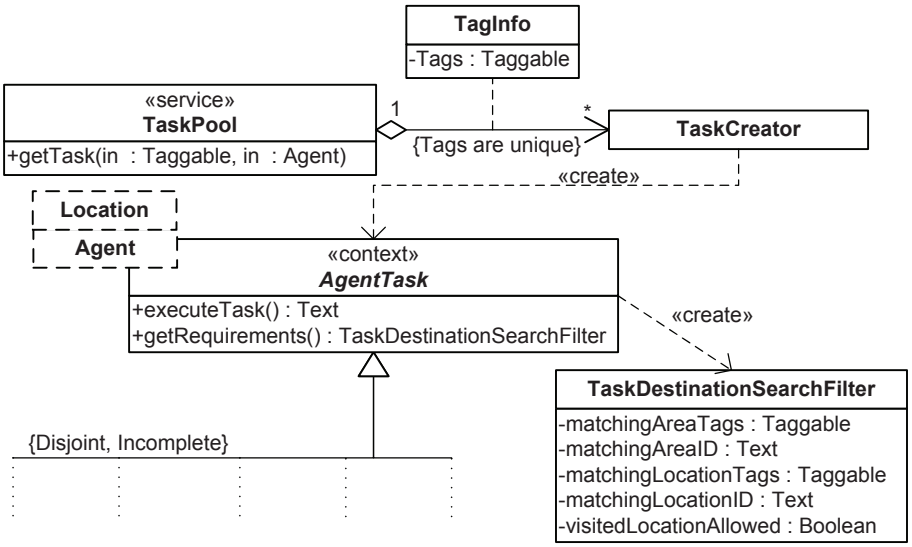


Figure 5.3: Contexts and interactions related to retrieving tasks in the agent architecture.

5.1.3 An Example of Using the Agent Architecture Tasks

This section continues example of Section 4.1.2. Figure 5.4 shows the tags for agent architecture infrastructure defined in Figure 4.2. This figure has been extended from Figure 4.2 by specifying which tags each entity has. Agent architecture will automatically assign the IDs for each entity, if needed, so they are not required to be explicitly defined. Typically, the decision of what kind of tags things have is design-time decision.

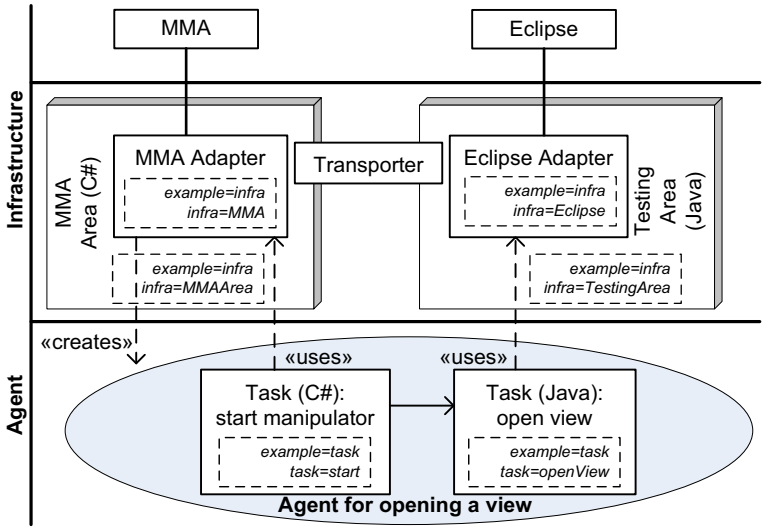


Figure 5.4: The agent architecture setup for the opening a view in Eclipse.

The tags are shown as a rectangle with dashed border line inside the boxes of the used areas, locations, and tasks. Inside the rectangle, each tag is on its own line.

The pattern for the text on each line is *key* = *value*. The tags of the Transporter are omitted from the picture for brevity.

Now that the tags has been decided, it is time to actually implement things. Let us assume that the MMA is written in `C#`, and the Eclipse plug-in is written in `Java`. Two methods shown in Listing 5.1 contain the code required to construct the main area of MMA. This code uses the *AreaFactory* and *TaskPool* services to create area and set up the task pool, respectively. The *AreaFactory* is part of abstract factory design pattern used to create Areas. These services are singletons and obtained at run-time using suitable frameworks of the native language.

```
public class MMASetup
{
  public Area CreateTAArea(AreaFactory areaFactory)
  {
    return areaFactory.CreateSubArea(new MMAAreaArchetype());
  }

  public void DoMMATaskPoolSetup(TaskPool taskPool)
  {
    taskPool.RegisterContextCreator(
      new DefaultTaskCreator(
        new StartApplicationTaskArchetype()
    ));
  }
}
```

Listing 5.1: Setting up the agent architecture environment for the sub-area of MMA.

The *MMAAreaArchetype* is a simple class implementing the required interface specified in agent architecture. This interface has methods to extract the information about what kind of tags, tasks, and locations the area will have. Thus building an area is a very simple action.

The *StartApplicationTaskArchetype* is similarly to *MMAAreaArchetype* information-encapsulating class about the starting application task. The task archetype provides information about the tags associated with the task, and the class of the task. Thus the *TaskPool* service will use the tags to locate the correct context creator, and the context creator will know which class to instantiate. This instantiated class will be the task to be executed. The code for both *MMAAreaArchetype* and *StartApplicationTaskArchetype* is given in Appendix 1.

The agent arhictecture-related task implementation to start an application can be seen in Listing 5.2. Each task must implement the AgentTask context interface defined by agent architecture. Additionally, agent architecture provides an *Agent-*

TaskSkeleton, a helper class with implementation for DCI-related things. This way task writer can concentrate on actual task. The *AgentTaskSkeleton* class also implements AgentTask but leaves those methods abstract, hence the *ShowDataTask* only extends *AgentTaskSkeleton* without needing to explicitly implement AgentTask.

```
public class StartApplicationTask : AgentTaskSkeleton
{
  public override TaskDestinationSearchFilter GetRequirements()
  {
    // Allocate the factory
    TaskDestinationFactory tdf = ...;

    return tdf.CreateFilter(
      null, // Any area tags are ok
      null, // Any location tags are ok
      null, // No need to match specific area ID
      null, // No need to match specific location ID
      true // Visited locations are allowed
    );
  }

  public override String ExecuteTask()
  {
    // Agent and Location are roles of AgentTask
    Agent agent = this.RoleMap.Get<Agent>();
    Location location = this.RoleMap.Get<Location>();

    // Start up application if needed
    ...

    // Return task result telling that all went fine
    return "TaskResultOK";
  }
}
```

Listing 5.2: Model implementation for task to start up application.

The *GetRequirements* method returns the filter to match locations against when searching the location to execute task. The task to start the application can be executed on any location, since it does not use any location methods. Therefore all task destination filter values are *null*. Abstract factory pattern is used once again to create *TaskDestinationSearchFilter* in order to express required location for task.

The *ExecuteTask* method will be invoked only when suitable location has been found for the task to execute on. Inside the method, it is shown how one gets hold of agent and location. Furthermore, it is visible how the task result is returned.

Listing 5.3 contains the code necessary to create an agent and start it up. Once again, the archetype is used to provide agent architecture with required information in order to create new agent. For this example functionality, the task graph is simple, as shown in Figure 5.4. There are two tasks to execute, and there is only one transition from first to second task. The the requirement of the transition that the output of the previous task is *TaskResultOK*. The agent archetype is given in Appendix 1.

```
public void OpenView(AgentFactory agentFactory,
  String manipulatorInfo, String applicationPath,
  Int32 viewID) {
  agentFactory.CreateAgent(
    new OpenViewAgentArchetype(
      manipulatorInfo, applicationPath, viewID)
  ).StartExecutingInNewThread();
}
```

Listing 5.3: Instantiation of agent for opening a view.

The agent archetype is given chance to add some data to its Data Container before the agent is returned to whoever wanted to create it. In this case, the name of manipulator, its application path, and the ID of the view are added to agent's Data Container. Once the agent has been created, it is started by either *StartExecutingInNewThread* or *StartExecutingInThisThread* methods. The first method creates designated thread to start executing the agent, and the second one blocks until this agent execution lifeline stops or the agent is transported elsewhere. No further actions are required from the user of the agent architecture. The framework will take care of the task resolution, the serialization and the deserialization, and the transportation of the agent.

Listing 5.4 and Listing 5.5 show the required code to set up sub-area of Eclipse plug-in and the implementation of the task to show data, respectively. Note that except for different syntax, code is very similar to `C#` version.

```
public class EclipseSetup
{
  public Area createTestingArea(AreaFactory areaFactory)
  {
    return areaFactory.createArea(new TestingAreaArchetype());
  }
```

```
  public void doEclipseTaskPoolSetup(TaskPool taskPool)
  {
    taskPool.registerContextCreator(
      new DefaultTaskCreator(
        new OpenViewTaskArchetype()
    ));
  }

  public void doEclipseLocationSetup(
    LocationMethodRetrieverService methods,
    EclipseManipulator manipulator )
  {
    methods.registerContextCreator(
      new DefaultLocationMethodContextCreator(
        new Tags( // Area tags
          new Tag("example", "infra"),
          new Tag("infra", "TestingArea")
        ),
        new Tags( // Location tags
          new Tag("example", "infra"),
          new Tag("infra", "Eclipse")
        ),
        manipulator // Resource
      )
    );
  }
```

Listing 5.4: Setting up the agent architecture environment for Testing sub-area.

```
public class OpenViewTask extends AgentTaskSkeleton
{
  public TaskDestinationSearchFilter getRequirements()
  {
    Agent agent = this.getRoleMap().get(Agent.class);
    String manipulatorInfo = agent.getDataContainer()
      .get("manipulator").toString();
    TaskDestinationFactory tdf = ...;

    if ("Eclipse".equals(manipulatorInfo))
    {
      return tdf.createFilter(
        new Tags( // Required tags for area
          new Tag("example", "infra"),
```

```
            new Tag("infra", "TestingArea")
          ),
          new Tags( // Required tags for location
            new Tag("example", "infra"),
            new Tag("infra", "Eclipse")
          ),
          null, // No need to match specific area ID
          null, // No need to match specific location ID
          true // Visited locations are allowed
        );
      } else { ... }
    }

    public String executeTask()
    {
      // Agent and Location are roles of AgentTask
      Agent agent = this.getRoleMap().get(Agent.class);
      Location location = this.getRoleMap().get(Location.class);

      LocationMethodRetrieverService locationMethods = ...;

      // Retrieve some native resource represented by this location
      EclipseManipulator manipulator = locationMethods
        .createContext(location)
        .getResource(EclipseManipulator.class);

      // Command manipulator to open view
      ...

      // Return task result telling that all went fine
      return "TaskResultOK";
    }
}
```

Listing 5.5: Model implementation for task to collect data in Eclipse plug-in.

As in C# version, archetypes are used to supply required information to agent architecture. All the archetypes of the Listing 5.4 are presented in the Appendix 1.

5.2 The Implementation of the Language-Independent Data Structures

5.2.1 The Data Content Definition of the Data Structures and Schemas

All the data structures defined in Section 4.3.2 are seen in Figure 5.5. The data structures specialize the common supertype LIObject. All hierarchical data structures further specialize the Composed, and atomic data structures specialize the Atomic. This kind of structure is just slightly more complicated version of composite pattern. The Composed is the abstraction of all the nodes, and the Atomic is the abstraction of all the leaves.

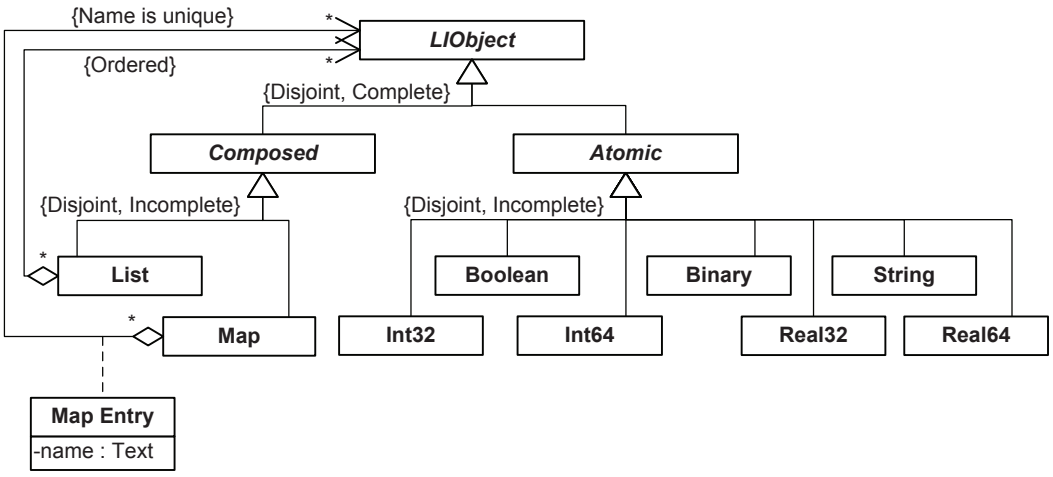


Figure 5.5: The language-independent data structures.

The inheritance hierarchies of Composed and Atomic are marked as *Incomplete* because these classes are *extension points*. The users of the agent architecture may introduce their own, customized, language-independent data structure types. This customization makes it easier to use the agent architecture in applications with a complex data model.

The concept of *schemas* has already been discussed in Section 2.2.4 and Section 4.3.2. The data model for schemas is seen in Figure 5.6. It imitates the schema data model in Apache Avro [28], but has fewer schema types. The functionality implemented in Avro as different schema types is partly implemented in Acceptor.

Each Schema has exactly one Acceptor, concrete subtypes of which may implement the **nil**-check, whereas this check is implemented in Avro as a special schema for **nil** objects. All of Schema, ComposedSchema, and Acceptor are extension points. Like with the language-independent data structures, the users of the agent architecture may add their custom schemas and Acceptors. Together with data structure

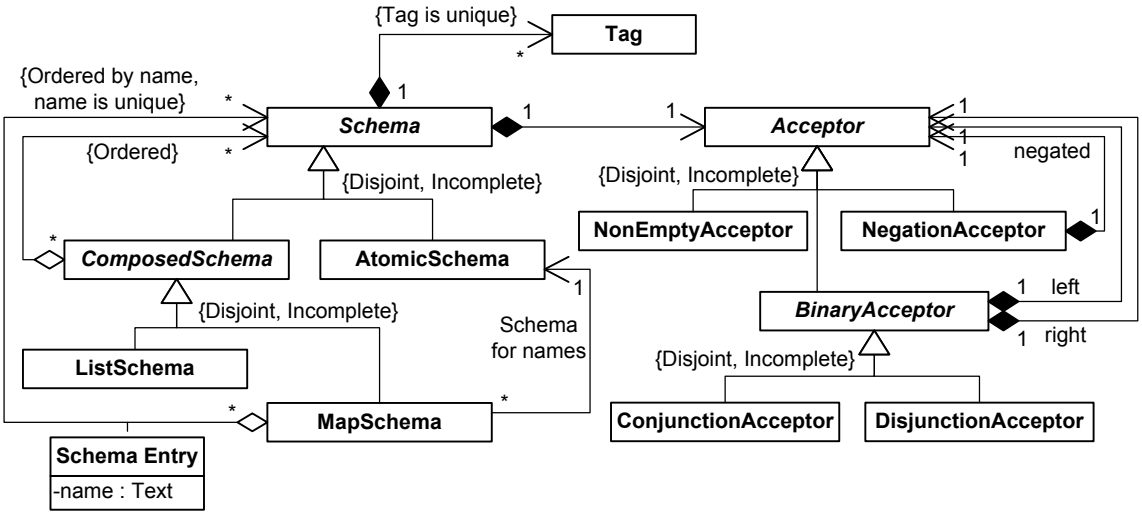


Figure 5.6: Data model for schemas to be used with the language-independent data structures.

extensibility, even the complex data models used in applications may be integrated seamlessly into the agent architecture. Once again, the composite pattern is visible in both schema and acceptor structures.

Additionally, each schema has zero or more Tags. These tags act as schema identifying mechanism. A schema with one or more tags may be referenced by just supplying tags, much like the naming mechanism in the Avro [28]. Whenever the actual schema behind this reference is required, the schema is retrieved from some persistence store using these tags as search criterion. This implies that *there cannot exist two schemas associated with same tags* in the persistence store.

5.2.2 Principle for Moving Native Objects Over a Network

Whenever there is a need to move some native object over a network, a certain principle is used. Using an agent as an example, the pattern is depicted in Figure 5.7.

First, the native agent object is *de-constructed* into language-independent data structures. During this phase, the formal definition for agent state, introduced in Section 4.2.1, is used to decide what data related to the agent to use, and which data structures to create. Then, these data structures are *serialized* into binary format, using a common schema for all agents. This schema, shown in Appendix 2, uses the formal definitions of Section 4.2.1 in order to create correct constraints for the data structures. Additionally, *validation* is performed during serialization, in order to see which data structures adhere to the provided schema. A more compact serialization format is used for the data structures adhering to the schema.

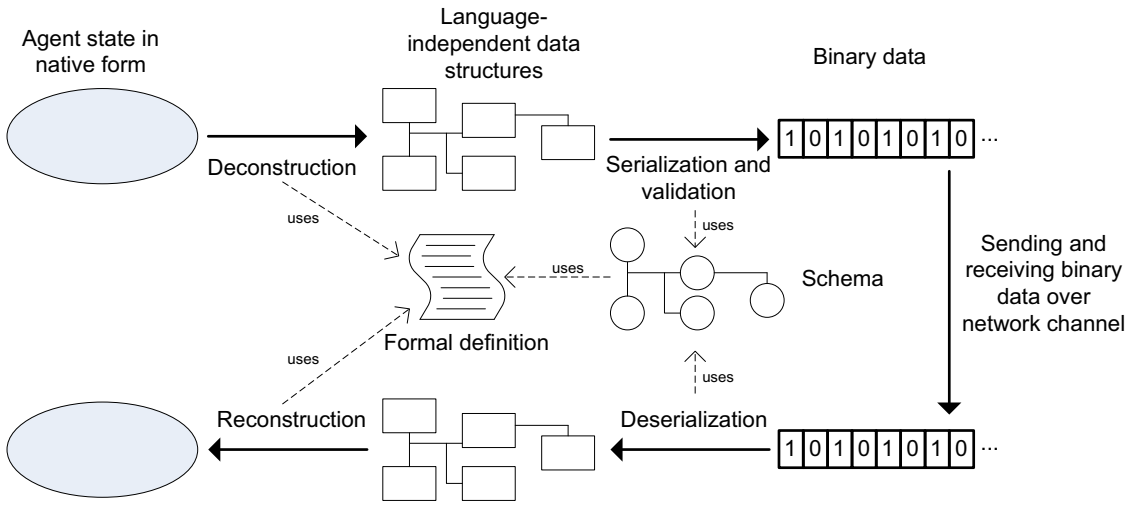


Figure 5.7: Principle for moving agents over a network using language-independent data structures.

On a receiving end, this process is reversed. First, the received binary data is *deserialized* into data structures using the schema for all agents. Then, the native agent object is *reconstructed* from these data structures.

5.2.3 Implementation of Deconstruction and Reconstruction

Before transmitting any language-independent data structures among applications, these data structures must be created. Since the data structures should be easy to create from existing structures, native to the programming language in use, the concepts of *deconstruction* and *reconstruction* are introduced. During deconstruction, a single language-independent data structure is created from a single native data structure. Consequentially, during reconstruction, a single native data structure is created from a single language-independent data structure.

Agent architecture uses DCI-pattern in order to automatize the deconstruction and reconstruction of Schemas and Acceptors. This is done because both of these types are extension points and thus the functionality of deconstruction and reconstruction for them should be easily extended. Figure 5.8 presents the contexts and required roles used for deconstruction and reconstruction of Schemas and Acceptors. These context creation services are not visible in the figure since they operate on the exactly same principle as services presented in Figure 5.2 and Figure 5.3. The only concrete difference is that context creators are mapped based on the type information. The correct context creator is looked up at runtime based on the type of the given Schema or Acceptor.

Additionally, Tags and Agents can be deconstructed and reconstructed, and the services to perform these processes are also visible in the Figure 5.8. However, since neither is an extension point, the usage of DCI-pattern is not required.

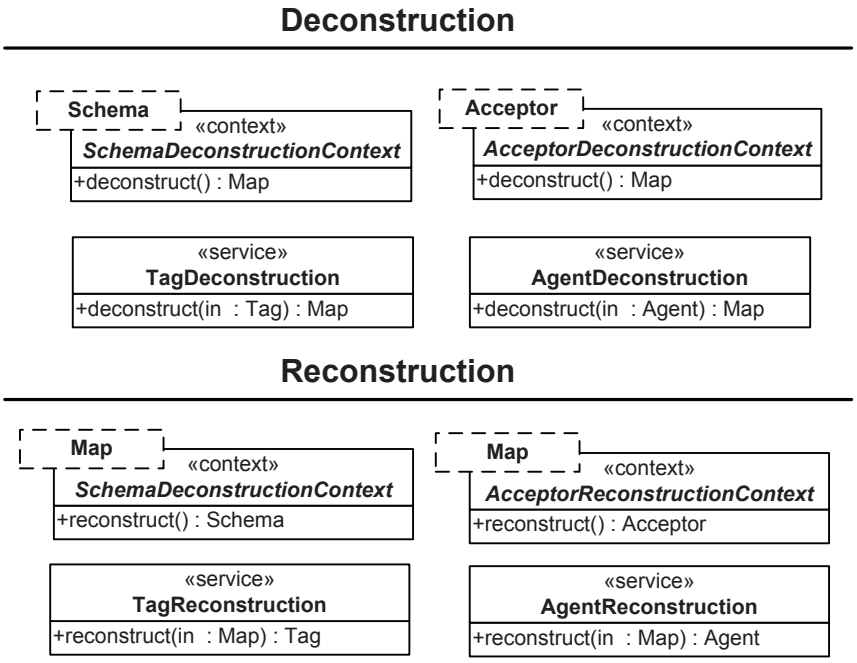


Figure 5.8: Contexts and services for deconstruction and reconstruction of schemas, acceptors, tags, and agents.

5.2.4 Implementation of Serialization and Deserialization

Now we have a customizable and automated way to transform native data structures into language-independent data structures. These language-independent data structures can be further used in *serialization* and *deserialization* processes. Figure 5.9 shows contexts and required roles for serialization and deserialization.

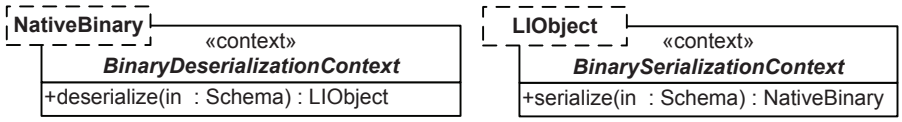


Figure 5.9: Contexts for serialization and deserialization of language-independent data structures.

In this case the usage of DCI-pattern is required because both extension points Composed and Atomic extend LIObject. Once again, the context creator lookup logic works same way as with deconstruction and reconstruction. Context creator is looked up based on runtime type information of the LIObject to be serialized or deserialized. The BinaryDeserializationContext has a native binary object in the role of NativeBinary as data source. A language-independent data structure is produced as a result, using the context creation service for deserialization to deserialize possibly contained serialized language-independent data structures. Correspondingly, the BinarySerializationContext does exactly the same thing in other direction. The Schema parameter in the methods *serialize* and *deserialize* is optional.

Presented serialization and deserialization processes can operate only on language-independent data structures. Coupled with deconstruction and reconstruction, this kind of mechanism is extremely extensible and robust. The logic to decompose and recompose native data structures is in one place, and the logic to serialize and deserialize the language-independent data structures is in another place.

Additionally, since the Schemas themselves are fully deconstructable and thus serializable, agent architectures may exchange the schemas between application boundaries. This enables easy solution for situations, where the Data Container of the transported agent contains LIObject serialized with Schema that is not present in the application receiving the agent. Then the receiving application just needs to ask the schema to be transported from the application from where agent arrived. The only problem will arise if the schema or acceptor are customized and their corresponding implementation is not present in the receiving application. Such situations however typically indicate an error in the design of the software environment.

5.2.5 Implementation of the Validation of the Data Structures

The serialization output can be further optimized by using some Schema during serialization and deserialization. This is shown in Figure 5.9, where the *serialize* and *deserialize* -methods both take schema as parameter. In order to detect whether some data structure adhers to certain schema, agent architecture has concept of *validation*. The validation provides information which data structure matches which schema, and it is typically used during serialization and deserialization of language-independent data structures. Figure 5.10 provides a detailed information on how the validation is implemented using DCI-pattern. This figure is more detailed than the previous ones because the validation is a little more complex functionality.

The singleton object, service ValidationService is used to start the validation process. First, the context is created for validation — the concrete type of this context will be decided by ValidationContextCreator. The service holds a mapping based on type information, and will use the runtime type of given Schema or Acceptor as a key to look up the context creator. The context creator will create the actual context with the implementation of its *validate* method corresponding the given Schema or Acceptor. This context will have given LIObject in the role of its type, and either Schema or Acceptor in the role of their type. The roles are then used by a context to determine whether the given LIObject adhers to the given Schema or Acceptor.

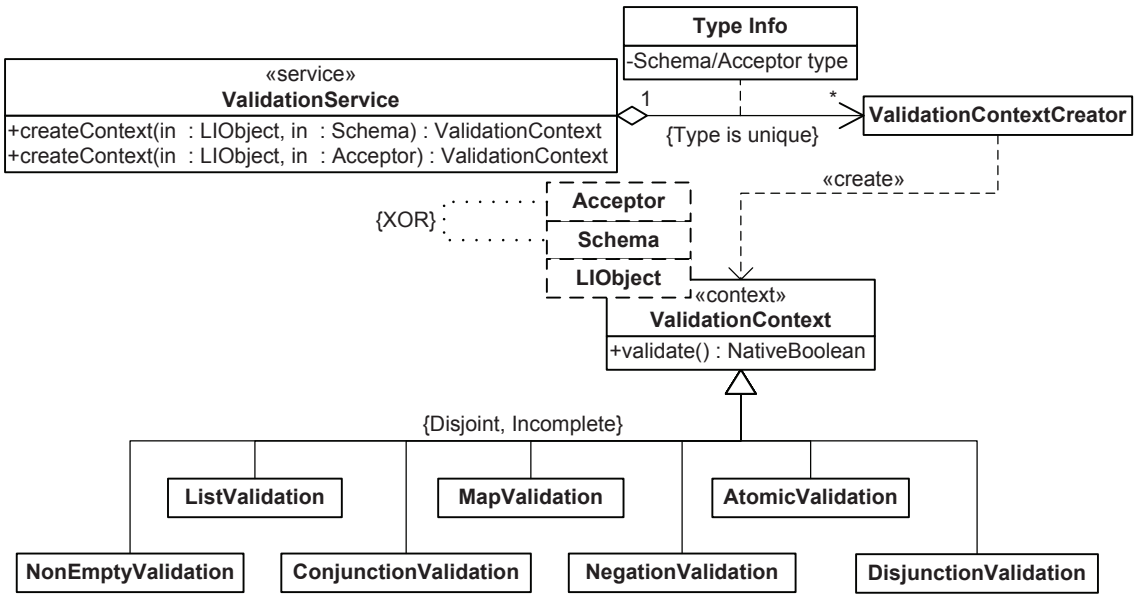


Figure 5.10: Context, context creation service, and context creator for validation process.

5.2.6 The Default Validation Functionality

The Figure 5.6 only has a few of concrete Acceptors defined in order to keep the figure simple. Table 5.1 lists all the default acceptor types. Each acceptor type is concrete type, and $Acceptor(a, o)$ means the behaviour of the acceptor's validation context *validate* method seen on Figure 5.10, where a is instance of the acceptor, and o is the data structure given to acceptor.

5.2.7 An Example of Using the Language-Independent Data Structure

This section continues working on the same example as in the Section 5.1.3. Now it is time to show the full code of the task executing methods for both `Java` and `C#` tasks. This code typically uses language-independent data structures when interacting with the Data Container of the agent. The task implementation for `C#` is shown in Listing 5.2, and for `Java` in Listing 5.5.

```
public override String ExecuteTask()
{
  // Agent and Location are roles of AgentTask
  Agent agent = this.RoleMap.Get<Agent>();
  Location location = this.RoleMap.Get<Location>();

  // Start up application if needed
  if (<application not running>)
  {
```

Table 5.1: Full feature list of the LID along with the explanations.

Acceptor type	$Acceptor(a, o)$
AcceptNonNil	o is Primitive and $value(o) \neq$ **nil**
AcceptNil	o is Primitive and $value(o) =$ **nil**
AlwaysAccept	**true**
ConjunctionAcceptor	$Acceptor(left(a), o) \wedge Acceptor(right(a), o)$
DisjunctionAcceptor	$Acceptor(left(a), o) \vee Acceptor(right(a), o)$
MandatoryNames	o is Map, and $\forall name \in names(a) : o$ has object with name $name$, where $names(a)$ returns all names associated with a
NegationAcceptor	$\neg Acceptor(negated(a), o)$
NeverAccept	**false**
NonEmpty	o is Composed, String, or Binary, and $length(o) > 0$, where $length(o)$ returns amount of contained data structures of o in case of Composed, or length of contents otherwise
OneOf	$\exists p : p \in objects(a) \wedge p = o$, where $objects(a)$ returns set of objects associated with a
RegexpAcceptor	o is String, and contents match the regular expression of the acceptor

```
    String path = agent.DataContainer
      .Get<StringObj>("applicationPath").StringValue;

    Process.Start(path); // Process is native .NET class
  }

  // Return task result telling that all went fine
  return "TaskResultOK";
}
```

Listing 5.6: Starting an application inside an agent task.

The creation of language-independent data structures uses abstract factory pattern. The code to detect whether the application is running is omitted for brevity, since the check involves some concurrency and security issues. Furthermore, the code assumes that archetype stored path to application, Eclipse in this case, by the name *applicationPath* in the Data Container of the agent.

```
public class OpenViewTask extends AgentTaskSkeleton
```

```
{
  public String executeTask()
  {
    // Agent and Location are roles of AgentTask
    Agent agent = this.getRoleMap().get(Agent.class);
    Location location = this.getRoleMap().get(Location.class);

    LocationMethodRetrieverService locationMethods = ...;

    // Retrieve some native resource represented by this location
    EclipseManipulator manipulator = locationMethods
      .createContext(location)
      .getResource(EclipseManipulator.class);

    // Command manipulator to open view
    int viewID = agent.getDataContainer()
      .get(Int32Obj.class, "viewID").getIntegerValue();
    manipulator.openView(viewID);

    // Return task result telling that all went fine
    return "TaskResultOK";
  }
}
```

Listing 5.7: Retrieving a 32-bit integer into agent's data container.

The `Java` code assumes that the agent archetype stored the ID of the view to open by the name *viewID* in the Data Container of the agent. The way to use LocationMethodRetrieverService is also presented in the `Java` code of Listing 5.7.

In case of any language, once the data is stored into data container, it will be automatically serialized by the agent architecture if agent is transported. No further effort is required from the user of the agent architecture. However, she may provide a schema to use when serializing some object in data container.

6. EVALUATION

6.1 Experiences Related to Case Study

Because of the prototype nature of Trinity, first positive experience related to agent architecture is that it actually works. There are currently agents successfully transported between MMA and Eclipse plug-in. The data and execution state of these agents are fully serialized before transportation, and successfully deserialized after the transportation. Additionally, the execution itself is resumed after transportation and is continued on target area successfully. The full schema for the agents is shown in Appendix 2.

As an example, Trinity's open view function was described in great detail in Sections 3.1.3, 4.1.2, 5.1.3, 5.2.7. It was shown that during the agent transportation, the deserialization process happens completely out of the sight of the user of the agent architecture. The data container is fully deserialized, and the tasks may ask it for data in the native, language-dependent way. Of course, when the user of the agent architecture wishes to extend its language-independent capabilities, she is required to directly interact with reconstruction, deconstruction, serialization, deserialization, and validation mechanisms. However, the data model and each mechanism can be easily and independently extended, easening up the learning curve and effort required to extend the language-independent capabilities of the agent architecture.

By having agent architecture plugged into each application in Trinity, the management of cross-application features becomes easy. The incremental development is well supported since each task is an independent item, and the task graph for each agent can be customizable at runtime. Additionally there are no limitations as to which task is usable in which agent, so each task may be reused in any agent.

The ability of agent architecture to abstract away all the details of language-independent way of moving both data and execution state has proven out to be invaluable. There is no need to worry about custom protocols or network channels, since that part is handled by the agent architecture. Instead, the users of the agent architecture may concentrate purely on domain-specific features to be implemented. Thus, a lot of work is saved.

As a downside, the agent architecture is still something one would not use for small applications or applications that do not interact with other applications. Because of language-independence, the way things are done is very indirect and may be

hard to follow by someone unfamiliar with the agent architecture. Also, the agent architecture needs a lot of configuration to be done before it is usable. This is partly because of the nature of the agent architecture. However, this makes the threshold to use the agent architecture higher.

6.2 General Observations

The problem of heterogeneous software application was solved by the language-independent agent architecture in an adequate way. The cross-application functionality was easy to maintain and extend. Furthermore, the users of the agent architecture did not need to worry about the details of working in heterogeneous software environment. These details include, but are not limited to, serializing domain-specific data in a language-independent way, looking up the task destination in the environment and determining whether transportation was required, and finally, running the agent itself and making it execute the required tasks.

The general security problem of mobile code is that it possesses a threat for the machine where it is executed. The code from remote machine may hijack the local machine and use it for some malicious purposes. This threat is not present in the implemented agent architecture, since the transported agents contain only *description* of the code to execute in form of the tags used for task resolution. Thus, we always know what code will be executed on local machine, and this security threat is eliminated.

This security problem did not, however, completely vanish. Instead, the implementers of the agent tasks must now trust completely that a task with certain tags will indeed do what it is supposed to do. However, this should not be a big problem in closed software environments.

The implementation of the agent architecture utilized very heavily the DCI pattern. Overall, the pattern has left positive impression. Both data model and features may be extended independently and easily, and the features may be customized at runtime, by providing a new context creator for some feature. Additionally, in most cases, the roles of interactions proved out to be just interfaces of data model. The only exceptions occurred when the roles needed to be some native resources, such as **NativeBinary** in Figure 5.9. As a downside, each time feature is used, some performance loss occurs as the lookup for context creator, and creation of the context object is performed.

6.3 Proposals for Improvement and Criticism

We briefly brief mentioned about task-level forks and joins in Section 4.2.2. Currently there is no implementation nor definitions as to how to implement parallelism

on task level. The biggest difficulty lies in defining the logic for merging the **Data Container** in join situations, and performing sync for other execution lifelines to complete the join. These definitions are left to be defined later. However, as a temporary workaround, these limitations can be overcome by creating agents with tasks to execute in parallel, and by waiting for their execution to finish.

The agent architecture is defined to be separately implemented for each programming language in Section 3.2.2. This is not a problem if the amount of the languages to support is small, like two or three. However, when this amount rises to higher numbers, the actual agent architecture implementation becomes fragmented.

In such situations, every decision changing something in the *concept* of agent architecture propagates into multiple changes in *implementations* of agent architecture. One possible solution would be creating some kind of customizable way of describing code of many programming languages in one place. Then agent architecture implementation for each language would be generated from this description, limiting the design-level changes into one place. Fortunately, the concept-level changes occur rarely.

The implementation-level changes to e.g. transportation format may occur more often, and they still require action to keep all agent architecture implementations up to date. However, these changes are invisible to the users of the agent architecture, as these implementation details are abstracted away.

Finally, even though performance has not been completely forgotten in the design and implementation of agent architecture, it has not been the major priority. Therefore it is required to perform an inspection into possible performance bottlenecks and extensive testing on whether these bottlenecks actually appear. It is not clear what are the situations when, for example, message-based approach is more performance-effective, and when it is less performance-effective. Therefore it would also be necessary to gain some comparable data on this issue.

Currently the agent architecture is implemented only in two languages: `Java` and `C#`. Because of very similar nature of these languages, it is not yet completely clear whether the same design- and implementation-specific definitions will work with more exotic programming languages. Therefore some effort should be made into implementing agent architecture in other languages. One interesting idea is to provide implementation in C++-language, as it lacks reflection completely, and some DCI-related services rely heavily on existence of at least rudimentary reflection mechanism.

7. CONCLUSIONS

In this thesis a language-independent agent-based architecture was defined. The language-independence support was achieved by first designing a very high-level description of the concepts of the agent architecture. Then the data state and execution state of the agent were *formally defined*. Finally, the specifications for implementations of the agent architecture were provided, aimed to maintain high level of extensibility and ease of use.

The design and definitions used in this agent architecture were verified by having a software environment composed of applications written in different languages. The cross-application features of this heterogeneous environment were successfully implemented and used. Additionally, this thesis examined one cross-application function of the use case very closely as an example. The agent architecture fulfilled well the requirements posed by being used as an integration architecture in heterogeneous software environment.

Despite having full language-independence aspect, this agent architecture is not too complicated to use. One way to achieve this was to make decision that there is be one agent architecture implementation for each programming language. Additionally, the logical design of the agent architecture was kept as simple as possible, letting the users of agent architecture to be able concentrate on domain-specific issues.

The agent architecture implementation for each language was also specified as strictly as possible. The key requirements for the implementations were the extensibility and ease of use. The requirements were fulfilled well thanks to the DCI pattern used in the implementation. The concepts of DCI aim to support the extensibility as much as possible, and the paradigm proved out to be worth the promises in implementations of agent architecture for `Java` and `C#`.

There are some features still left undefined, such as task-level parallelism of the agent. As future work, a detailed performance analysis is required. Additionally, implementation for more exotic or low-level programming languages is aimed to be provided. Thus, the language-independence of design and implementation specifications of agent architecture will be more thoroughly verified.

REFERENCES

[1] Encoding - Protocol Buffers - Google Code. Retrieved on May 26th, 2011, from http://code.google.com/apis/protocolbuffers/docs/encoding.html#types

[2] MIDI File Format: Variable Quantities. Retrieved on May 26th, 2011, from http://ctrlr.org/midi/midifile/vari.htm

[3] Tclers Wiki. Retrieved on July 3rd, 2011, from http://wiki.tcl.tk/

[4] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985 (1985)

[5] Altheim, M., McCarron, S., Ishikawa, M.: XHTML™ 1.1 - module-based XHTML - second edition. W3C recommendation, W3C (Nov 2010), http://www.w3.org/TR/2010/REC-xhtml11-20101123/

[6] Baumann, J., Hohl, F., Rothermel, K., Straßer, M.: Mole – Concepts of a mobile agent system. World Wide Web 1(3), 123–137 (Sep 1998)

[7] Öberg, R.: Contexts are the new objects. Retrieved on June 30th, 2011, from http://www.jroller.com/rickard/entry/contexts_are_the_new_objects

[8] Carzaniga, A., Picco, G.P., Vigna, G.: Is code still moving around? looking back at a decade of code mobility. In: Companion to the proceedings of the 29th International Conference on Software Engineering. pp. 9–20. ICSE COMPANION '07, IEEE Computer Society, Washington, DC, USA (2007)

[9] Cucurull, J., Martí, R., Navarro-Arribas, G., Robles, S., Borrell, J.: Full mobile agent interoperability in an IEEE-FIPA context. J. Syst. Softw. 82, 1927–1940 (December 2009), http://portal.acm.org/citation.cfm?id=1645443.1645578

[10] Cugola, G., Ghezzi, C., Picco, G.P., Vigna, G.: A characterization of mobility and state distribution in mobile code languages. In: Proc. of the 2nd ECOOP Workshop on Mobile Object Systems (Jul 1996)

[11] ECMA: ECMA-262: ECMAScript language specification. Tech. rep. (Dec 2009), http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[12] Franklin, S., Graesser, A.: Is it an agent, or just a program?: A taxonomy for autonomous agents. In: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages. Springer-Verlag (1996)

[13] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)

[14] Gerber, A., Raymond, K.: MOF to EMF: there and back again. In: Proceedings of the 2003 OOPSLA workshop on Eclipse Technology eXchange. pp. 60–64. Eclipse '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/965660.965673

[15] Gray, R.S., Cybenko, G., Kotz, D., Peterson, R.A., Rus, D.: D'agents: applications and performance of a mobile-agent system. Softw. Pract. Exper. 32, 543–573 (May 2002)

[16] Johansen, D., Lauvset, K.J., van Renesse, R., Schneider, F.B., Sudmann, N.P., Jacobsen, K.: A TACOMA retrospective. Softw. Pract. Exper. 32, 605–619 (May 2002), http://portal.acm.org/citation.cfm?id=586287.586293

[17] Koskimies, K., Mikkonen, T.: Ohjelmistoarkkitehtuurit. Talentum Media Oy (2005)

[18] Lange, D.B., Oshima, M.: Seven good reasons for mobile agents. Commun. ACM 42(3), 88–89 (1999)

[19] Muhametsin, S., Vartiala, M., Peltonen, J.: A model for language independent mobile agents. In: In Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11). pp. 90–101. TUT Press (2011)

[20] Object Management Group: Unified Modeling Language. Tech. rep. (May 2010), http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

[21] Overeinder, B.J., Brazier, F.M.T.: Scalable middleware environment for agent-based internet applications. In: In Proceedings of the Workshop on State-of-the-Art in Scientific Computing (PARA'04). pp. 675–679. Springer (2004)

[22] Peine, H.: Application and programming experience with the Ara mobile agent system. Softw. Pract. Exper. 32, 515–541 (May 2002), http://portal.acm.org/citation.cfm?id=586287.586290

[23] Peltonen, J., Vartiala, M.: An agent based architecture style for application integration. Annales Univ. Sci. Budapest., Sect. Comp. 31, 3–22 (2009)

[24] Postel, J.: Transmission Control Protocol. RFC 793 (Standard) (Sep 1981), http://www.ietf.org/rfc/rfc793.txt, updated by RFCs 1122, 3168, 6093

[25] Reenskaug, T., Coplien, J.O.: The DCI Architecture: A New Vision of Object-Oriented Programming. Retrieved on June 30th, 2011, from http://www.artima.com/articles/dci_vision.html (Mar 2009)

[26] Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift: Scalable Cross-Language Services Implementation. Retrieved on May 25th, 2011, from http://thrift.apache.org/static/thrift-20070401.pdf

[27] Sperberg-McQueen, C.M., Yergeau, F., Maler, E., Paoli, J., Bray, T.: Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C (Nov 2008), http://www.w3.org/TR/2008/REC-xml-20081126/

[28] The Apache Software Foundation: Apache Avro. Retrieved on May 4th, 2010, from http://avro.apache.org/

[29] The Unicode Consortium: The Unicode Standard, Version 6.0.0. The Unicode Consortium (2011), http://www.unicode.org/versions/Unicode6.0.0/

[30] Vartiala, M.: Design and implementation of an agent-based architecture for a process support system. Master's thesis, Tampere University of Technology (Mar 2010)

APPENDIX 1: THE CODE FOR ARCHETYPES

```
public class MMAAreaArchetype : AreaArchetype
{
  public MMAAreaArchetype()
  {
    this
      .withAreaTags(new Tags(new Tag("example", "infra"),
                             new Tag("infra", "MMAArea")))
      .withLocationArchetypes(
       new LocationArchetype().withTags(
         new Tags(new Tag("example", "infra"),
                  new Tag("infra", "MMA"))),
       new LocationArchetype().withTags(
         new Tags(new Tag("example", "infra"),
                  new Tag("infra", "Transport")))
       )
      .withTasks(new Tags(new Tag("example", "task"),
                          new Tag("example", "start")));
    }
}
public class StartApplicationTaskArchetype : TaskArchetype
{
  public StartApplicationTaskArchetype()
  {
    this
      .withTags(new Tags(new Tag("example", "task"),
                         new Tag("example", "start"))
      .withType(typeof(StartApplicationTask));
  }
}
public class OpenViewAgentArchetype : AgentArchetype
{
  private readonly String _manipulatorName;
  private readonly String _applicationPath;
  private readonly Int32 _viewID;
  public OpenViewAgentArchetype(String manipulatorName,
    String applicationPath, Int32 viewID)
  {
    this._manipulatorName = manipulatorName;
    this._applicationPath = applicationPath;
```

```
    this._viewID = viewID;
    this
      .withTask(new Tags(new Tag("example", "task"),
                         new Tag("task", "start")))
      .withTask(new Tags(new Tag("example", "task"),
                         new Tag("task", "openView")))
      .withTransition(1, "TaskResultOK", 2);
  }
  public override void PopulateDataContainer(Agent agent)
  {
    LIObjectFactory lif = ...;
    agent.DataContainer.Set("applicationPath",
      lif.CreateString(this._applicationPath));
    agent.DataContainer.Set("manipulator",
      lif.CreateString(this._manipulatorName));
    agent.DataContainer.Set("viewID",
      lif.CreateInt32(this._viewID));
  }
}
```

The code for archetypes to be used in MMA.

```
public class TestingAreaArchetype extends AreaArchetype
{
  public TestingAreaArchetype()
  {
    this
      .withAreaTags(new Tags(new Tag("example", "infra"),
                             new Tag("infra", "TestingArea")))
      .withLocationArchetypes(
        new LocationArchetype().withTags(
          new Tags(new Tag("example", "infra"),
                   new Tag("infra", "Eclipse"))),
        new LocationArchetype().withTags(
          new Tags(new Tag("example", "infra"),
                   new Tag("infra", "Transport")))
        )
      .withTasks(new Tags(new Tag("example", "task"),
                          new Tag("task", "openView")));
    }
}
public class OpenViewTaskArchetype extends TaskArchetype
{
```

```
  public StartApplicationTaskArchetype()
  {
    this
      .withTags(new Tags(new Tag("example", "task"),
                         new Tag("task", "openView"))
      .withClass(StartApplicationTask.class);
  }
}
```

The code for archetypes to be used in Eclipse plug-in.

APPENDIX 2: THE SCHEMA FOR AGENTS

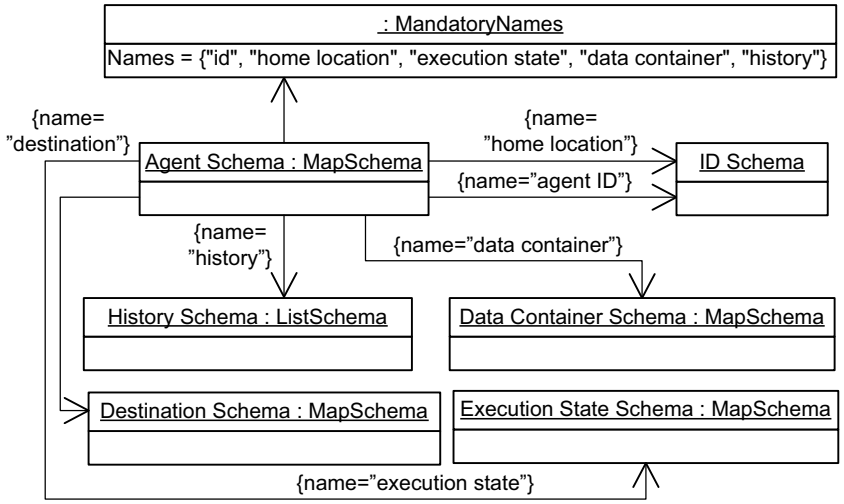


Figure A2.1: The schema for agents in the agent architecture.

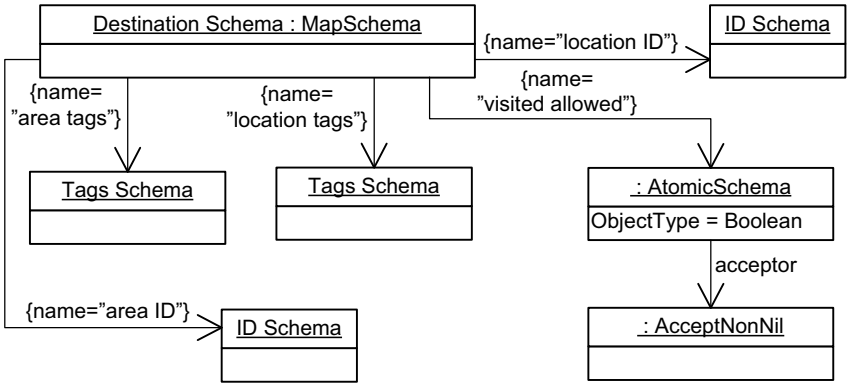


Figure A2.2: The schema for agent destination in the agent architecture.

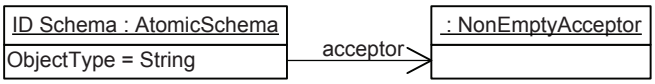


Figure A2.3: The schema for IDs in the agent architecture.

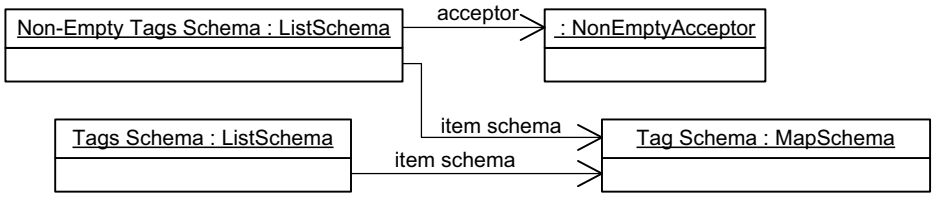


Figure A2.4: The schema for tags in the agent architecture.

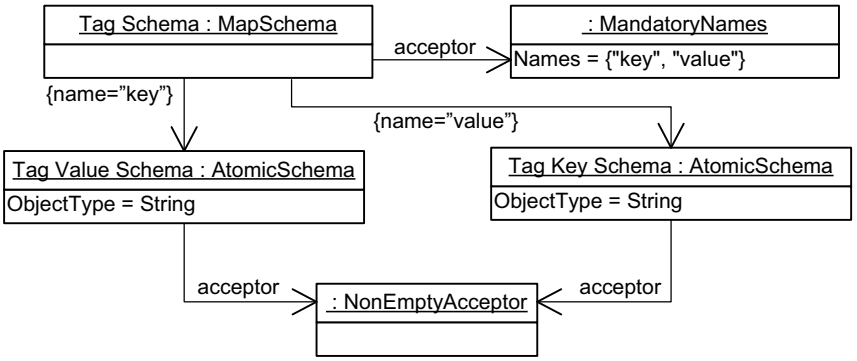


Figure A2.5: The schema for a single tag in the agent architecture.

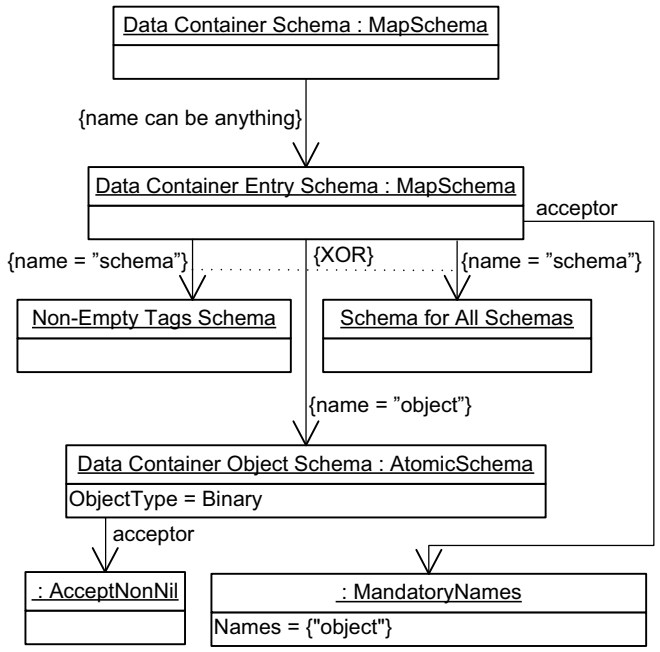


Figure A2.6: The schema for data containers in the agent architecture.

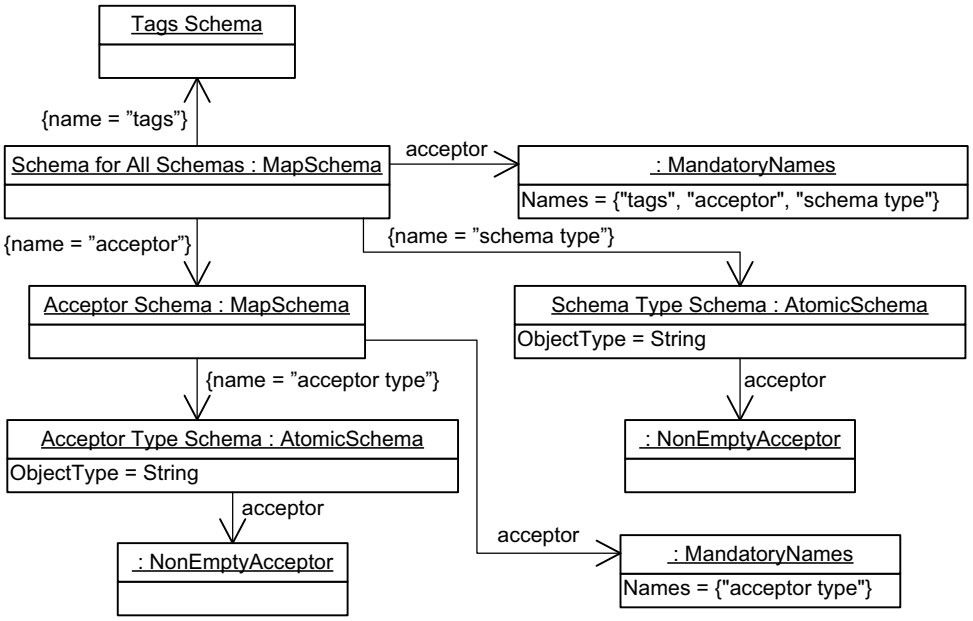


Figure A2.7: The schema for all schemas in the agent architecture.

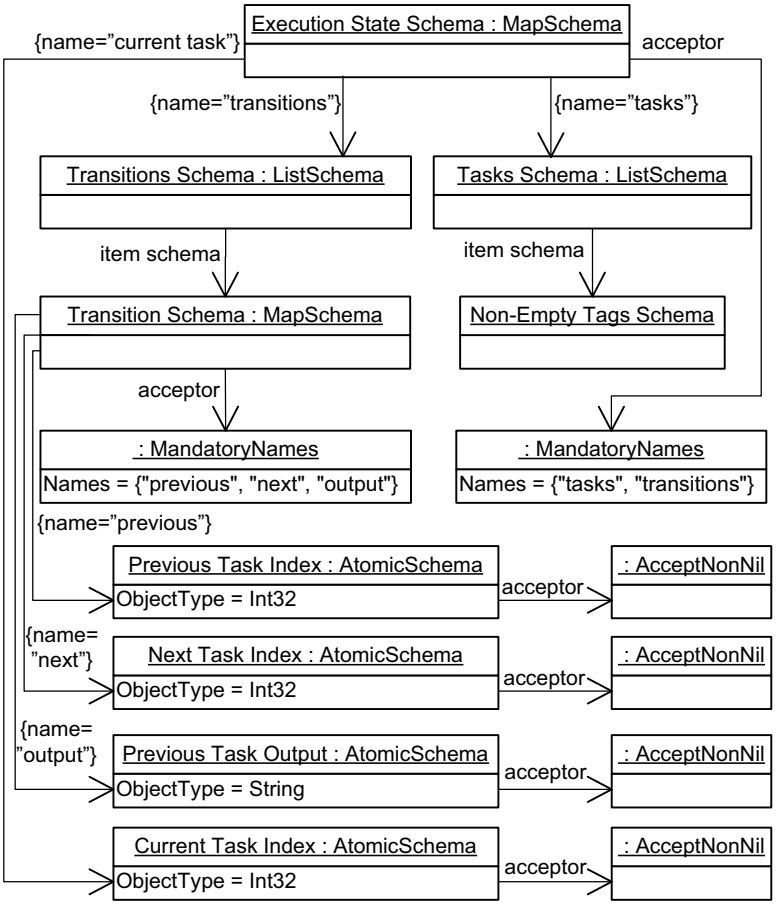


Figure A2.8: The schema for agent execution states in the agent architecture.

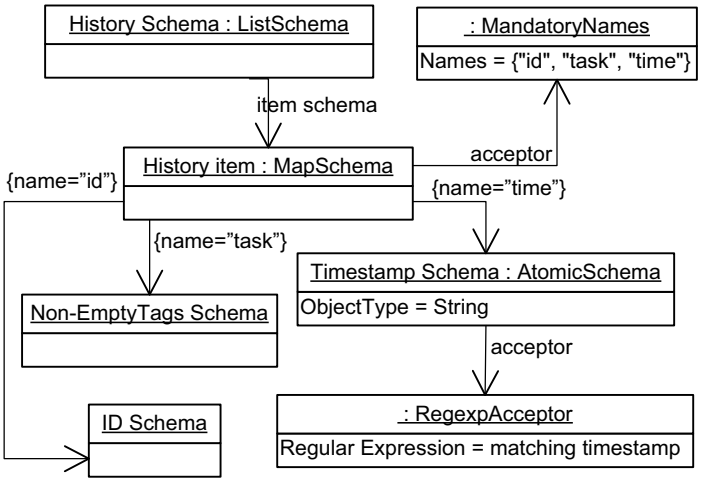


Figure A2.9: The schema for agent history information in the agent architecture.